DOCUMENT RESUME

ED 134 229 IR 004 441

AUTHOR	Nievergelt, J.; And Others
TITLE	ACSES: The Automated Computer Science Education System at the University of Illinois.
INSTITUTION	Illinois Univ., Urbana. Dept. of Computer Science.
SPONS AGENCY	National Science Foundation, Washington, D.C.
REPORT NO	UIUCDCS-R-76-810
PUB DATE	Aug 76
GRANT	EC41511; EPP-74-21590
NOTE	171p.
EDRS PRICE	MF-$0.83 HC-$8.69 Plus Postage.
DESCRIPTORS	Artificial Intelligence; College Curriculum; *Computer Assisted Instruction; *Computer Science Education; Information Retrieval; *Instructional Innovation; Instructional Systems; On Line Systems; Programing Languages
IDENTIFIERS	PLATO IV

ABSTRACT

The Automated Computer Science Educational System (ACSES) has been developed at the University of Illinois for the purpose of providing improved education for the large number of students taking introductory computer science courses. The major components of this system are: a library of instructional lessons, an interactive programing system with excellent error diagnostics, an information retrieval system, an automated exam and quiz system, and several lessons which judge student programs. This report briefly describes each of these components, as well as some ideas on programing language design resulting from this experience, and presents an evaluation of the use of the system over the past three years. (Author)

****
* Documents acquired by ERIC include many informal unpublished *
* materials not available from other sources. ERIC makes every effort *
* to obtain the best copy available. Nevertheless, items of marginal *
* reproducibility are often encountered and this affects the quality *
* of the microfiche and hardcopy reproductions ERIC makes available *
* via the ERIC Document Reproduction Service (EDRS). EDRS is not *
* responsible for the quality of the original document. Reproductions *
* supplied by EDRS are the best that can be made from the original. *
****

ED134229

ACSES: The Automated Computer Science Educatio
System at the University of Illinois

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY.

by

Authors:

J. Nievergelt
H. G. Friedman, Jr.
W. J. Hansen
R. G. Montanelli, Jr.
T. R. Wilcox
R. L. Danielson
R. I. Anderson
A. M. Davis
D. R. Eland
D. W. Embley
W. D. Gillett
P. Mateti
J. L. Pradels
E. R. Steinberg
M. H. Tindall
L. R. Whitlock

"PERMISSION TO REPRODUCE THIS COPYRIGHTED MATERIAL HAS BEEN GRANTED BY

J. Nievergelt

TO ERIC AND ORGANIZATIONS OPERATING UNDER AGREEMENTS WITH THE NATIONAL INSTITUTE OF EDUCATION. FURTHER REPRODUCTION OUTSIDE THE ERIC SYSTEM REQUIRES PERMISSION OF THE COPYRIGHT OWNER."

August 1976

IR004444

This work was supported in part by the National Science Foundation under grants EC41511 and EPP-74-21590.

Acknowledgements

We are indebted to a host of students who have worked on portions of ACSES as term projects or Master's theses, as well as those who participated in the initial use of the system for instruction.

The assistance and advice of the staff of the PLATO IV project have been appreciated, as well as the support provided by the National Science Foundation.

Finally, thanks are due to Betsy Colgan for an excellent job of typing (and sometimes retyping) this report.

TABLE OF CONTENTS

Page

1. Introduction (J. Nievergelt) 1

2. The library of lessons (H. G. Friedman, Jr.) 10

3. Computer assisted programming system (CAPS) (T. R. Wilcox, A. M. Davis, M. H. Tindall) 12

4. The GUIDE information and advising system (D. R. Eland, J. L. Pradels) 45

5. Interactive test construction and administration in the generative exam system (L. R. Whitlock, R. I. Anderson) 63

6. Automatic judging of student programs (R. L. Danielson, P. Mateti, W. D. Gillett) 73

7. Experimental and formal language design applied to control constructs for interactive computing (D. W. Embley) 103

8. Use of ACSES in instruction (R. G. Montanelli, Jr., E. R. Steinberg) 112

9. ACSES bibliography 141

Appendix: Computer Science Lessons 144

1. Introduction (J. Nievergelt)

From 1972 to 76 the Department of Computer Science has been heavily involved in a project to develop an automated instructional system for teaching computer programming. After four years of implementation, with an effort in excess of 25 man-years which produced approximately a million words of code, our system ACSES (Automated Computer Science Education System) is now in routine large-scale use, assuming about 50% of the teaching load in various introductory computer science courses, with a total enrollment of over 1500 students per semester.

ACSES runs on the PLATO IV system developed by the Computer-based Education Research Laboratory at the University of Illinois. The approximately 1000 terminals across the country attached to the Illinois PLATO system have permitted a smaller scale use of ACSES in some other schools. It is to be expected that the use of ACSES will continue to increase, in our own courses as well as elsewhere. While the initial development of ACSES is now complete, expanded use requires a continuing effort to maintain the system: adapting it to changes and new features of the PLATO system, adding new instructional material, and, most important, improving existing material on the basis of experience in actual instructional use.

The purpose of this report is to document the ACSES project: to serve as a case study in the design, implementation, and use of a major effort in computer-aided instruction. This introduction is a concise description of the ACSES project: it presents the motivation for starting the project, the design criteria, the components of the resulting system, the experience gained during the implementation and use of ACSES. The remainder of the report describes various aspects of the project in more detail.

Why ACSES?

Computers are playing an increasingly pervasive role in our society, and this fact leads directly to a rapidly increasing demand for basic computer science education. This demand arises from two sources.

First, the demand for computer professionals continues to grow. The most widely accepted projections show a doubling of the total demand for systems analysts, programmers, computer operators and associated technicians during the next five years.

Secondly, it seems reasonable to expect that most people will be required to interact with computers in their daily work within a decade or two. Even people not directly concerned with computers should have some understanding of them because an enlightened public will be essential if we are to make intelligent decisions concerning the future role of computers in our society. Currently many citizens view computers with indifference, while others fear them as the ultimate threat to their privacy, security, and dignity. Such attitudes will clearly not suffice. Hence, it is important that every educated person have some understanding of the principles underlying computers and their implications for society, as well as some skill in their use. At the very least, everyone should have the opportunity to acquire this knowledge in a convenient way.

Recognizing the importance of "computer literacy," the President's Science Advisory Committee recommended in their 1967 report that 75% of all college students should have a meaningful exposure to computers. Even though this percentage has not yet been attained, the demand for instruction in basic computer science at our universities, colleges, junior colleges, and private electronic data processing schools is enormous. These institutions have relied on the traditional instructional approach of lecture-discussion-laboratory. This approach suffers from several defects, particularly when it involves large numbers of students. It is not particularly suited to the subject and is, therefore, the cause of student dissatisfaction. Learning to program requires active participation and intense effort on the part of the student--two things that are not encouraged by the lecture type of instruction. An individual tutor for every student would be ideal for learning a skill such as programming, but such a mode of instruction is obviously economically not feasible when one aims at a mass-education program.

In addition, the traditional lecture-based approach can only reach a limited audience. In particular, it excludes all people whose professional duties prevent them from attending school for any length of time. It is to be expected that there will be a large demand for basic computer science courses in connection with continuing adult education programs. The situation where somebody suddenly finds that he should know something about computers in order to remain effective on his job, will become an increasingly familiar event.

All of these long-range considerations, in addition to our own experience in teaching introductory computer science at the University of Illinois to about 2000 students of widely different backgrounds every semester, led to the initiation of a large-scale project to automate introductory computer science courses, the results of which are described in this report.

The PLATO IV system being developed by the Computer-based Education Research Laboratory at the University of Illinois gave us a unique opportunity to develop an automated course consisting of CAI-lessons about computers and of supporting software, and to try this system out on a large audience of diverse educational background. The PLATO IV system, while centered at the University of Illinois, serves nearly 1000 terminals located at schools and colleges with very different types of student populations, and with wide geographic dispersion.

Potentially, computer-assisted instruction (CAI) has many advantages: It can provide truly individualized instruction by allowing students to study sequences of lessons tailored to their needs and at their own pace; given a suitable terminal network, it can reach a wide audience and, in the forseeable future, do so at low cost. Because of its potential cost effectiveness, CAI may become the cheapest way for schools and colleges who do not as yet offer computer science courses to institute programs in computer studies. However, these potential advantages of CAI will be realized only if much more research and a large scale development effort is carried out. We view our project as a contribution towards the goal of demonstrating the feasibility of a CAI-based approach to the problem of mass-education in basic computer science.

Design criteria, and the resulting structure of ACSES

ACSES, our automated computer science education system developed on PLATO IV, is designed to be usable in two modes, according to the two purposes it is intended to serve: supplementary instruction at our own university, and main-line instruction at remote sites.

a) The partially automated mode for supplementary instruction

In our Computer Science Department an instructor is still responsible for a course, and CAI lessons are used in an adjunct mode. He may discuss a problem in classroom, and then refer the students to an appropriate lesson that gives more detail, examples, or allows the student to practice or solve problems on the computer. In the case of our computer science lessons, practicing and problem solving usually means that the student must write and execute a small program. He is able to do so at the same terminal, and switch easily from lesson taking to programming, and back.

Thus, in order to operate a partially automated introductory computer science course, one needs primarily:

- -- a library of lessons, covering several programming languages, computing techniques, and application areas
- -- a completely self-contained interactive programming system for the preparation, execution and debugging of programs written by students in any of the languages covered by the lessons.
- -- an exam system, to automatically generate problems according to an instructor's specification, to grade the student's solution, and administer the exam (data collection and security aspects).

b) the fully automated mode for main-line instruction

We expect that the demand for basic computer science courses on the part of high schools, junior colleges, and continuing adult education will grow rapidly in the near future. In these settings, there may not be an instructor available who can guide the student's course of study and fill any gaps that might be present in the lesson material. Hence in this setting the system has to be usable in a fully automated mode, and ACSES makes this possible primarily by providing:

-- a conversational advice-giving and information retrieval system to guide the student through the library of lessons, based on his goals and past performance

-- a communication system that allows a student to contact a human tutor for help and advice, from any terminal connected to the PLATO system.

We consider our project to be a major research effort to investigate the extent to which a large introductory course can be automated. The objective has not been to take one of our existing CS courses and put it on PLATO as it is. Rather, we want to turn a PLATO terminal into a rich environment, analogous to a conventional library and laboratory, where you have at your fingertips many useful things for learning about computer science, and for practicing immediately what you have learned. It is the student and his instructor who decide which one of these things they want to use.

Also, the project has an additional goal; namely to serve as a stimulating environment for computer science research in a variety of areas: compilers, information systems, artificial intelligence. This last point may deserve some explanation, since the misconception is widespread that lesson writing is a routine activity. It can be, if one creates poor lessons. It can also be a task as challenging as you wish to make it, if you view a lesson as an interactive program that has a certain domain of knowledge, and is able to communicate with students about the knowledge it has. We feel that if this necessary ingenuity and effort are put into the design of an instructional system and its supporting software, such a system can be made to provide an educational experience superior to the one a student has in a large introductory course taught in the conventional lecture-based manner.

Experience gained during the implementation

From the technical point of view, the design and implementation of a computer-based instructional system is no different than the development of a large system for some other application. Conventional software components dominate: compilers, interpreters, filing systems, data management. A high level programming language is desirable, almost a necessity. Tutor, the only programming language available to developers of instructional material on PLATO, is a very high level language for programming man-machine dialogs, but is rudimentary with respect to structuring large software systems into self-contained modules, and designing clean interfaces between them. It is well suited to programming small or medium-sized (a few thousand words of source or object code) conventional lessons, which are I/O intensive and, as programs, have a simple structure. It is considerably less well suited for writing large, complex programs, such as the compiler system, the information and advising system, and the exam system of ACSES.

Another hindrance which affects the performance of our complex programs, in particular the interactive compilers, is that PLATO limits one user's program to 2 msec of CPU time per clock second under normal system loads. This is sufficient for the conventional lessons, but it causes the response time in more complex programs to be irritatingly slow.

With respect to educational and psychological aspects of a computer-based instructional system, it must be said that there is no systematic body of knowledge to guide the designer of such a system. The voluminous literature on CAI, to the extent that it reports on experiments to determine what are effective environments for learning, treats detailed aspects in isolation, under controlled conditions which do not apply to the varied situations that occur when an entire course is given by computer. The best advice is to try everything in actual instruction as soon as possible, to be prepared to make major modifications in response to feedback from the users, and to discard unsuccessful material. The most productive point of view seems to be to consider a computer-driven graphics terminal as a medium with novel properties, and to develop a crafts of writing interactive programs for communication of all kinds, educational or other.

Experience using ACSES

Repeated experiments, questionnaires, and plain observation makes it clear that a large majority of the students like the experience of working on PLATO, and that many prefer studying a lesson to attending a lecture in the large classes typical of introductory courses. This is the most conclusive finding of our effort to evaluate ACSES in instruction. In contrast to this, we have no conclusive data for comparing the performance of students off and on PLATO - usually no difference between the two groups were detected. A common-sense conclusion is that other factors, such as the instructor, and the motivation of students, have a much stronger influence on the student's performance than the difference between a lecture and a CAI session.

Conclusion

The ACSES project can serve as a benchmark to indicate what effort is required to develop, maintain, and use a computer-based instructional system capable of assuming a large part of the teaching load in large introductory courses. We hope that others may benefit from the experiences described in this report.

2. The library of lessons (H. G. Friedman, Jr.)

The largest component of the ACSES system is our library of instructional lessons. We have over 135 individual lessons, grouped into about 20 areas (Appendix 1). The majority of these lessons have been created by students as term projects in courses on computer-assisted instruction, or as other academic activities, such as theses. As a result of this origin, our lessons exhibit a wide variance in level of quality.

A typical lesson in our library is a program which presents some information in graphical and textual form, and provides some opportunity for the student to practice and assess his understanding of the subject matter by solving problems and answering questions. It is comparable in scope to a tutorial article or short chapter in a book, and may occupy a student for half an hour to a few hours.

According to this analogy, our library of 135 lessons represents a small publishing enterprise. The effort required in creating and maintaining a collection of interactive programs for instruction (courseware), however, is significantly larger than it is for a comparable amount of material in textual form. The main reason why courseware development is so labor-intensive is that, on a medium as powerful as a computer with a graphics terminal, one wants to use techniques of presentation and interaction that are much more elaborate than anything that could be done on paper. Implementation of these techniques takes a lot of programming time. Moreover, lessons using elaborate techniques must often be revised several times, because little experience is available for guiding one on how to use these techniques effectively at first attempt.

This need for repeated revision based on experience with actual instruction, and the significant amount of effort required for each revision, are the causes for the fact that only a fraction of our lessons are anywhere close to their "final form". As might be expected, those lessons which have been used most in our own instruction, particularly the FORTRAN sequence, are the most polished.

Among the improvements to the lessons has been a certain amount of standardization. For example, the function of the keys that allow a student to control his path through a lesson has been standardized among all CS lessons, so that a student who has gone through a few lessons has become familiar with most of the control options in all lessons and can proceed through subsequent lessons more easily. These conventions are described in lesson csauthors; standard pieces of code, character sets, and micro tables have been collected in lesson cslibrary; other coding suggestions have been collected in lesson cscode.

Also, several communication lessons have been completed. Notefile csnotes serves a "bulletin board" function between authors. Lesson cscomments serves to record remarks made by students taking instructional lessons, for feedback to the authors of the lessons. Lesson cstalk allows real-time communication between an instructor and each of several students; this allows human assistance to a student by an instructor who might be located at a different PLATO site.

Finally, H. G. Friedman, who manages the library of lessons, has written a router lesson, csrouter, which allows students access to the library and maintains records on which lessons the student has entered, which lessons have been completed (as set by each individual lesson), elapsed time spent in each lesson, etc.

3. Computer assisted programming system (CAPS) (T. R. Wilcox, A. M. Davis, M. H. Tindall)

3.1. Introduction

The principle design goal of CAPS is to be able to diagnose student programming errors and help the student understand them. Unlike batch systems such as PL/C [3] or SP/k [5], CAPS does not attempt to "recover" from detected errors. Instead, it interacts with the student to inform him of his error and attempts to prompt him with suggestions about ways of correcting the error. The student is required to actually analyze the situation and then repair his program. Automatic diagnosis is provided both for compile-time errors and run-time errors.

Most components of CAPS are table driven. This permits a compact implementation and provides a convenient mechanism for supporting the different languages that are taught at the University. Tables for FORTRAN, PL/1 and COBOL have been written and put into classroom use. Tables for LISP, PASCAL, BASIC and SNOBOL are under preparation. Severe time and space constraints imposed by PLATO IV (as it services 500 terminals concurrently) limit the capabilities of CAPS to simple programs using only subsets of the above programming languages. CAPS is sufficient to handle the computing requirements for the greater part of a first semester programming course. Subset number five of SP/k [5], for example, is typical of the language features supported by CAPS.

3.2. System organization

The principle modules of the system are a program editor, a syntactic and static semantic error diagnostician, an interpreter for each language supported, a run-time error analyzer, a user program file manager, and a system's table builder and file manager (Figure 1).

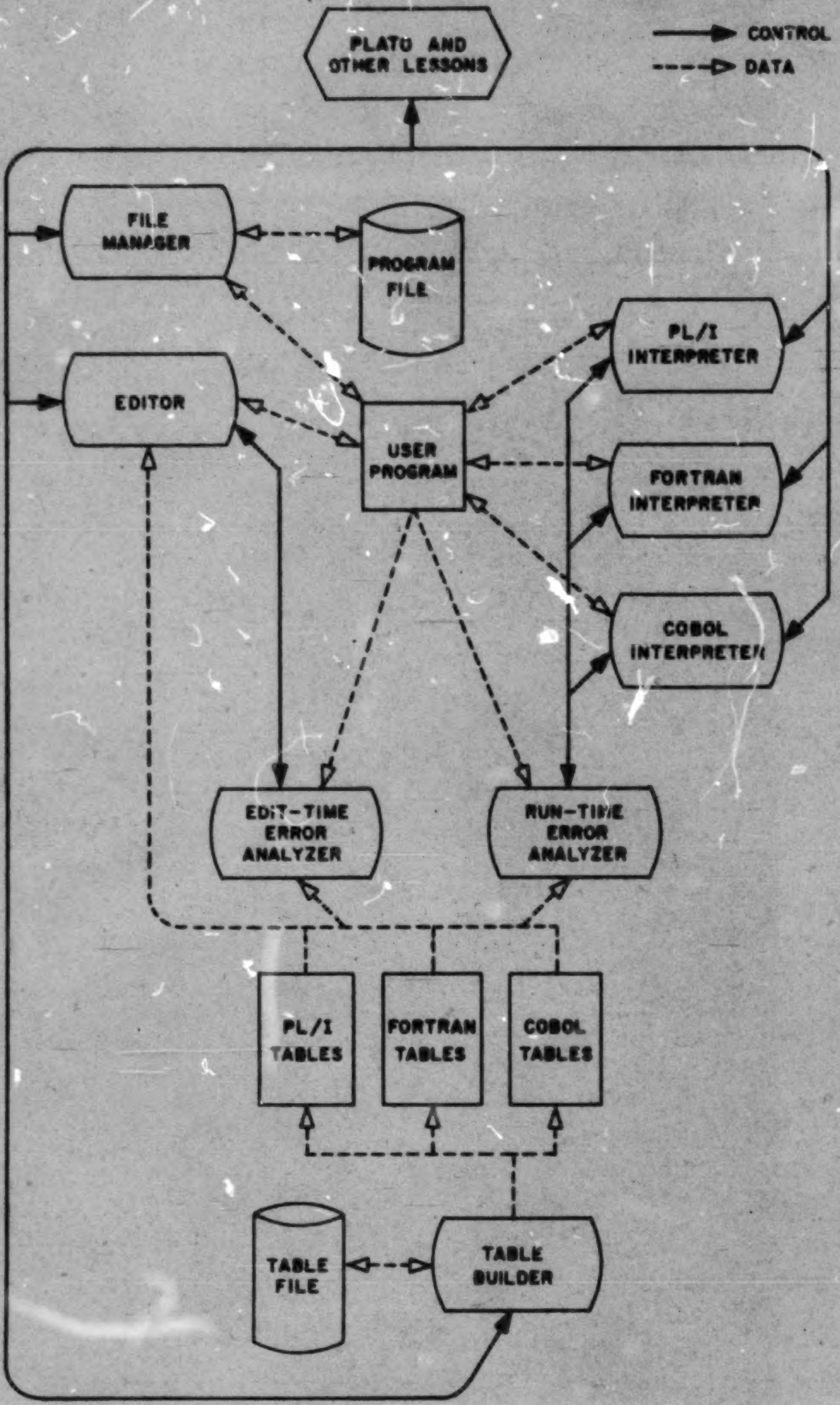


Figure 1: CAPS System Organization

Each of these modules is written in TUTOR - a FORTRAN level programming language and the only one used on PLATO IV [10] - and holds the position of a lesson within the PLATO IV system. All lessons are potentially reentrant pure procedures and they are the unit of multiprogramming in the PLATO IV system.

Control of the system is distributed throughout the modules, but the student is never aware of the system modularity and never has to remember command syntax, because each time the system is ready for a command, the module that will interpret the command displays a menu of possible actions. Usually pressing one key will initiate a new action.

During a programming session student-specific communciation between modules is through the block of storage (1500 words) assigned to each terminal. This block contains the only internal representation of the student's program, the associated symbol table entries and miscellaneous information identifying the student and the language he is using. The internal representation of the student's program is used both to regenerate the listing on the screen and as the "object code" for the language interpreter. It is a simple tokenization of the input. Between sessions only programs explicitly saved in the user program file are retained.

3.2.1. Program editor

Each editor in the CAPS system consists of interpretive tables specific to the language being compiled; common driving routines to interpret these tables; and a few routines, specific to the language, that are called from the interpreted tables. These tables are built

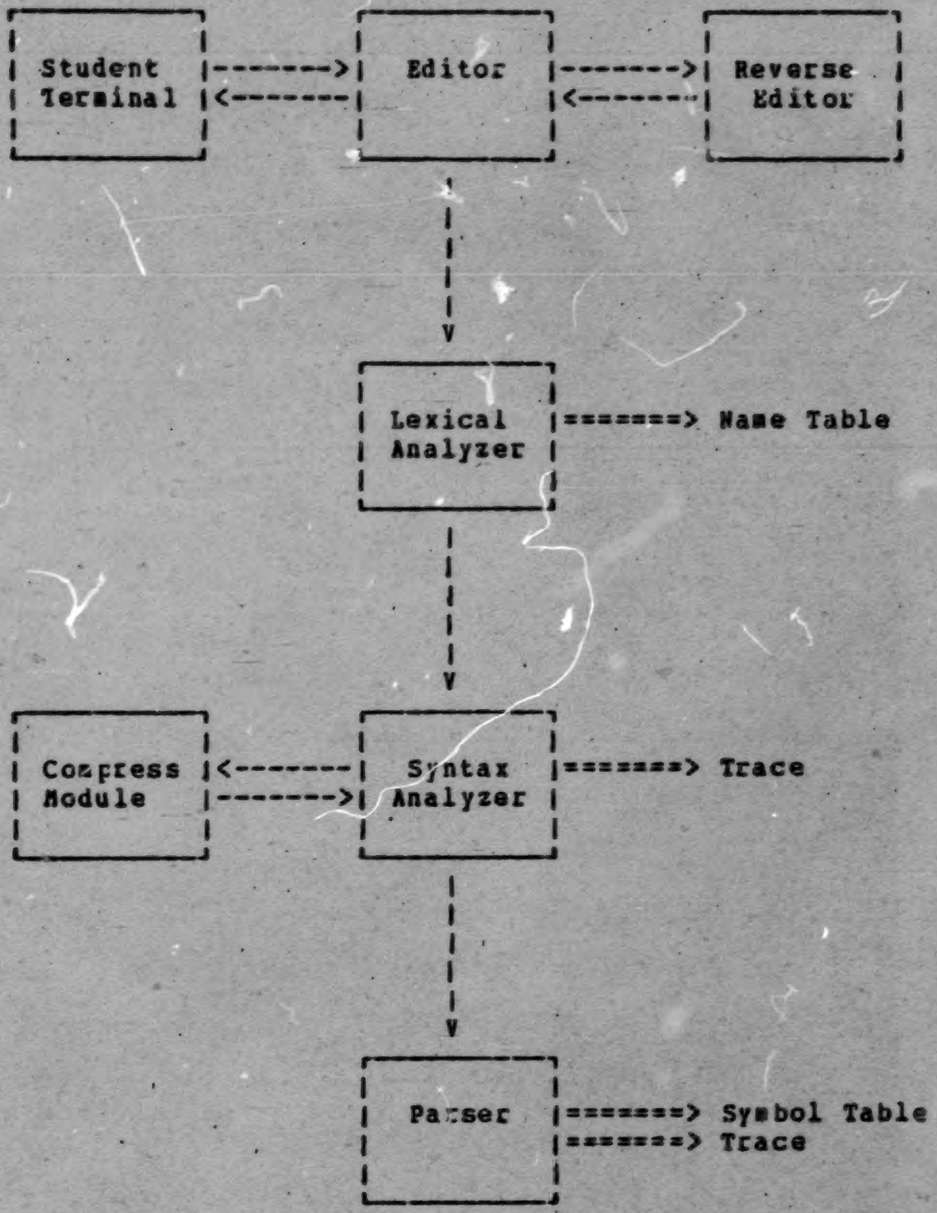


Figure 2a: CAPS Compiler Modules

from assembler-like source code written by a compiler implementor. After generation, these tables are stored in common where they are loaded into *nc* variables as needed in compiling student programs.

Flow of control in the CAPS compilers is shown in Figure 2a. The editor looks at each keypress the student enters from the terminal. If the key indicates a text editing function, it is performed by the editor. If the student is entering new text, each keypress is passed on to the lexical analyzer. When the lexical analyzer receives a complete token, that token is passed on to the syntax analyzer and parser for compilation. Since each keypress is processed as it is entered, the compiler can give immediate error messages when the student enters an invalid language construct.

While compiling new text, "Trace" information is stored, allowing the Reverse Editor to uncompile the student's program as the student backs up to make a change. Occasionally the storage area for Trace information gets full. When this happens, a compression unit is called which removes alternate entries from the Trace table. After compression is performed, the reverse editor can only back up to alternate tokens. If necessary, it will back up to the previous token and then forward compile to the current token. In practice, the compression routine may be called three or four times for a student program. After four calls, there is Trace information for one out of every 16 of the first tokens entered. Closer to the "cursor" where the student is working, the Trace information is available for every token, or at least alternate tokens.

The module labeled "syntax analyzer" in Figure 2a is just an interface between the lexical analyzer and the parser. Its function is to keep track of trace information and to insure that the correct tables are loaded for each routine.

3.2.2. Editor tables

Both the scanner and the parser are table driven. The scanner's table is a state transition matrix derived from the regular grammar that defines the tokens of the programming language. Special entries in the table are provided for handling fixed-format languages and continuation "cards". (See Wilcox [14] for more details.)

The parser is a recursive descent parser with one important difference: a single recursive procedure can be written to recognize the instance of more than one non-terminal. The non-terminal that has been found by such a procedure is passed back to the calling procedure where it can be used to choose the next alternative. (This is equivalent to the separable transition diagrams of Conway [2] or Tixier's [13] RCF languages). Lomet [7] has shown that such a system of procedures can recognize all deterministic context free languages.

Tindall [11] has designed a language for writing the procedures of the parser. The language provides assembly-like commands for obtaining and testing input symbols and performing simple symbol table operations. More complex semantic actions are carried out by TUTOR procedures called from the parser.

All of the tables for the editor can be constructed interactively using the table building modules. While under development the tables are stored on a disk file maintained by the table builders. For "public" versions of each language, the tables are assembled into condensed form and stored in a common data area to be shared by all terminals editing programs written in that language. With this system, editor and language development can proceed independently of editor use by students.

The CAPS compilers use "common" for pointers and tables shared by all users of one compiler and "storage" for all pointers and tables needed by an individual user. Few, if any, of the student variables are used by the compilers. Portions of common and storage may be loaded into 1500 *nc* variables in central memory. However, at most three areas of each may be loaded at once. As shown in Figure 2b, by arranging the data areas in ECS carefully, it was possible to meet this three-area restriction and still get the tables in desired locations in central memory. However, the lexical and parse tables are each 400 words long, and only one of them can be loaded at once. This is significant, since the compilers spend 8% of their time changing the loading arrangement. Figure 2b shows the layout of these areas.

3.2.3. Edit time error

The editor signals an error to the student by flashing a box around the invalid character or symbol (See Figure 3a). While the box is flashing, the editor monitors the student's key presses. Those which would move the cursor beyond the point of error are ignored; the other's are processed normally. When one of the keys moves the cursor back from the point of error, the box is erased and normal editing is resumed.

A special key, labeled HELP, provides the student with automatically generated diagnostic assistance. The diagnostician

```
           CM                                    ECS

  nc variables (1500)                Storage          (644)
 -------------------------------    -------------------------------
|                               |  |    Parse Storage 53           |
|    Lexical                    |  |-------------------------------|
|      or                       |  |v   Symbol Table 109           |
|    Parse                      |  |-------------------------------|
|    Tables                     |  |v   Name Table     64          |
|                               |  |-------------------------------|
|    400                        |  |v   Char Table    168          |
|                               |  |                               |
|-------------------------------|  |-------------------------------|
|    Parse Storage 53           |  |    Hash Table     20          |
|-------------------------------|  |-------------------------------|
|v   Symbol Table 109           |  |    Text           60          |
|-------------------------------|  |-------------------------------|
|c   Symbol Table 210           |  |    Trace          98          |
|                               |  |-------------------------------|
|-------------------------------|  |    Variables      88          |
|c   Name Table    110          |   -------------------------------
|-------------------------------|
|v   Name Table     64          |    Common          (1288)
|-------------------------------|   -------------------------------
|v   Char Table    168          |  |                               |
|                               |  |    Parse                      |
|-------------------------------|  |    Table                      |
|c   Char Table    119          |  |                               |
|                               |  |    400                        |
|-------------------------------|  |                               |
|    Hash Table     20          |  |                               |
|-------------------------------|  |-------------------------------|
|    Text           60          |  |                               |
|-------------------------------|  |    Lexical                    |
|    Trace          98          |  |    Table                      |
|-------------------------------|  |                               |
|    Variables      88          |  |    400                        |
 -------------------------------   |                               |
                                   |-------------------------------|
                                   |    Pointers       22          |
                                   |-------------------------------|
                                   |c   Symbol Table 210           |
                                   |                               |
 v = variable portion              |                               |
 c = constant portion              |-------------------------------|
 number = length of table          |c   Name Table    110          |
                                   |-------------------------------|
                                   |c   Char Table    119          |
                                    -------------------------------
```

Figure 2b: CAPS Editor Data Areas

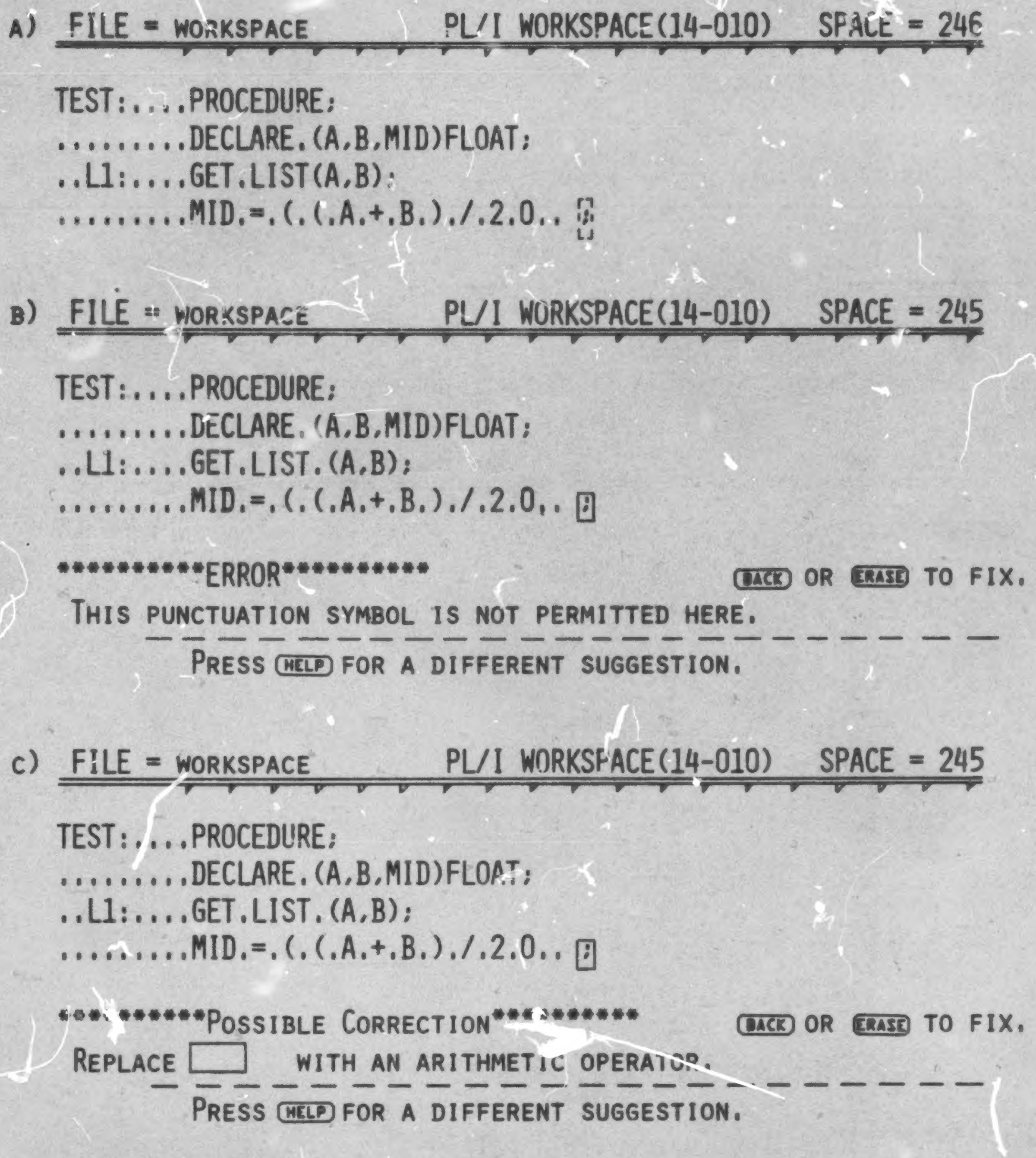
invoked by this key uses an algorithm similar to one proposed by Levy [6], but modified and improved by Tindall [12] for use in the interactive environment. The actions of the diagnostician will now be explained. These actions are illustrated in the continued example in Figure 3.

The first message generated by the diagnostician announces the type of the input symbol (e.g., keyword, arithmetic operator, etc.) and states simply that in the given context it is not permitted by the language (Figure 3b). The symbol class is derived from a field in the symbol table. The language tables include a mapping from values found in this class field into a suitable English phrase to be used in this and other messages.

After this information has been displayed, each time the student presses the HELP key again, he is shown a modification he could make to the text that would make the initial portion of his program legal. These suggested modifications are generated by attempting to insert, delete or replace a single symbol somewhere in the prefix of the sentential form that had been constructed by the parser up to the point of error. Only those modifications that lead to a successful parse all the way to the cursor are reported to the user (Figure 3c-31).

Modifications are attempted starting at the cursor and then working back toward the beginning of the program, one symbol at a time. Using this algorithm, each successive modification takes more time to compute, but the probability of a successful parse is less. The most likely modifications will be suggested first.

As the error handler works its way back from the cursor it backs up the parser so that it is always ready to accept input from the



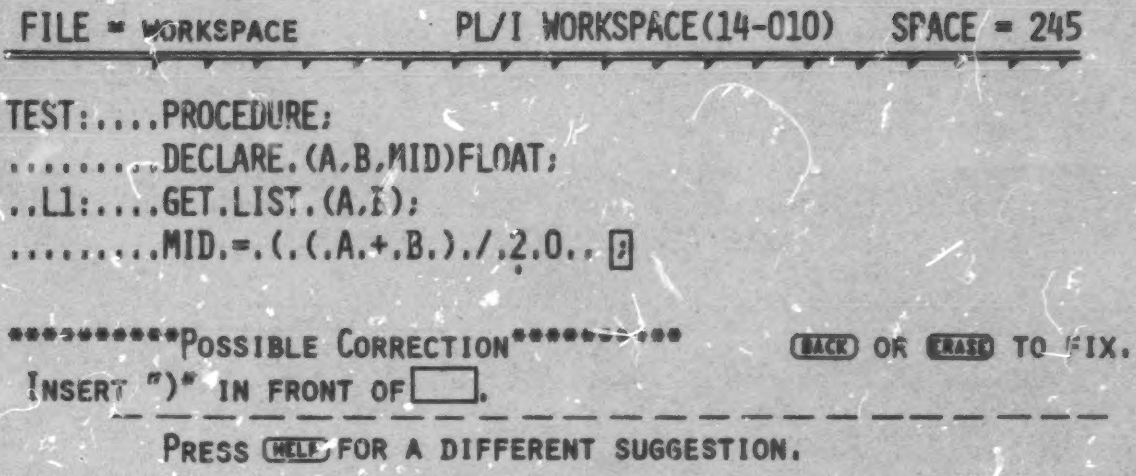
```
A)  FILE = WORKSPACE            PL/I WORKSPACE(14-010)   SPACE = 246

    TEST:....PROCEDURE;
    .........DECLARE.(A,B,MID)FLOAT;
    ..L1:....GET.LIST(A,B);
    .........MID.=.(.(.A.+.B.)./.2.0.. ▯
```

```
B)  FILE = WORKSPACE            PL/I WORKSPACE(14-010)   SPACE = 245

    TEST:....PROCEDURE;
    .........DECLARE.(A,B,MID)FLOAT;
    ..L1:....GET.LIST.(A,B);
    .........MID.=.(.(.A.+.B.)./.2.0,. ▯

    **********ERROR**********                    BACK OR ERASE TO FIX.
     THIS PUNCTUATION SYMBOL IS NOT PERMITTED HERE.
            - - - - - - - - - - - - - - - - - - - - - - - - - -
            PRESS HELP FOR A DIFFERENT SUGGESTION.
```

```
C)  FILE = WORKSPACE            PL/I WORKSPACE(14-010)   SPACE = 245

    TEST:....PROCEDURE;
    .........DECLARE.(A,B,MID)FLOAT;
    ..L1:....GET.LIST.(A,B);
    .........MID.=.(.(.A.+.B.)./.2.0.. ▯

    **********POSSIBLE CORRECTION**********      BACK OR ERASE TO FIX.
     REPLACE ▭    WITH AN ARITHMETIC OPERATOR.
            - - - - - - - - - - - - - - - - - - - - - - - - - -
            PRESS HELP FOR A DIFFERENT SUGGESTION.
```

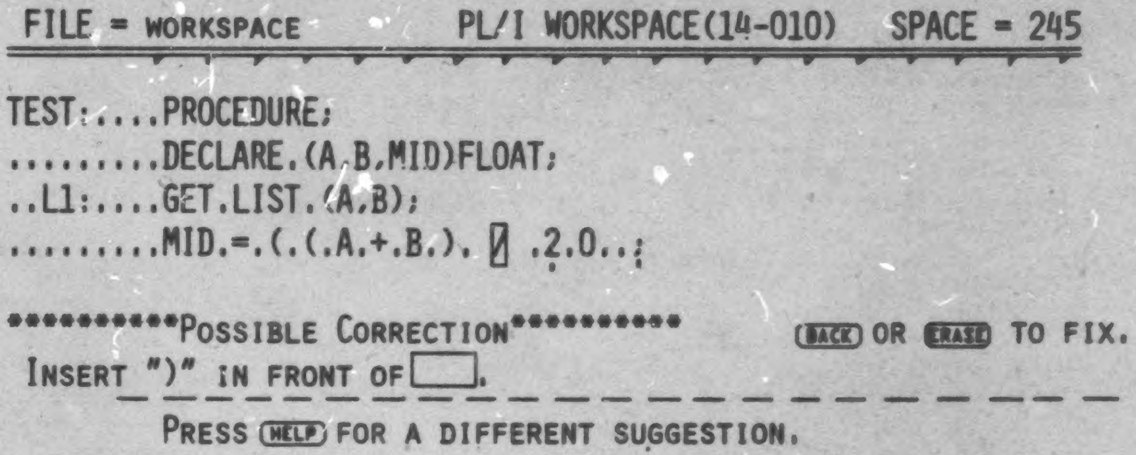
FIGURE 3. EDIT-TIME ERROR ANALYSIS

D)



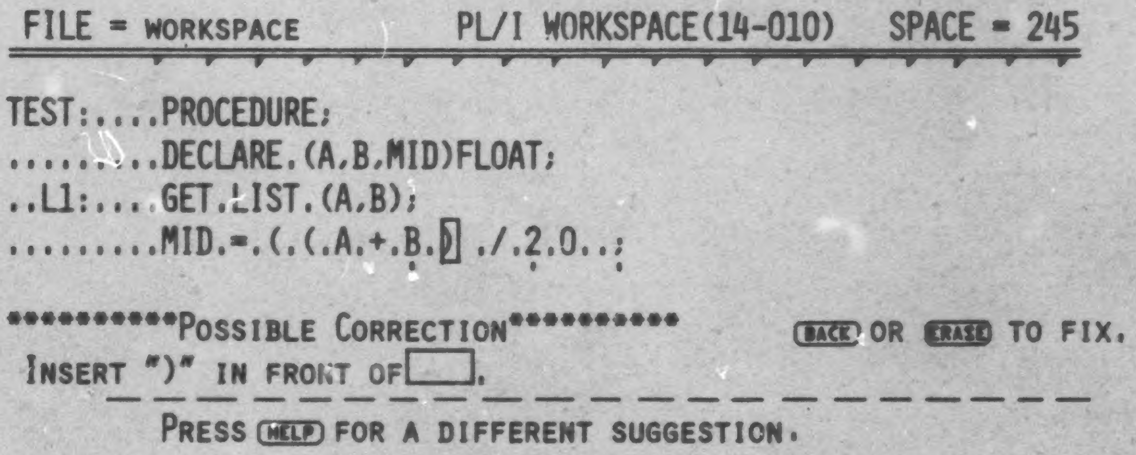
```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(A,B);
.........MID.=.(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********      BACK OR ERASE TO FIX.
 INSERT ")" IN FRONT OF    .
         PRESS HELP FOR A DIFFERENT SUGGESTION.
```

E)



```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(A,B);
.........MID.=.(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********      BACK OR ERASE TO FIX.
 INSERT ")" IN FRONT OF    .
         PRESS HELP FOR A DIFFERENT SUGGESTION.
```

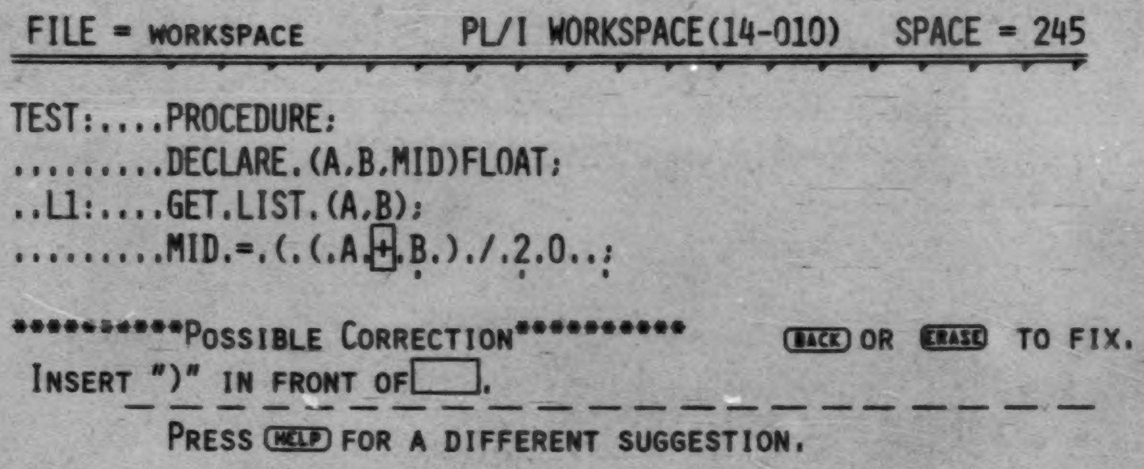
F)



```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(A,B);
.........MID.=.(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********      BACK OR ERASE TO FIX.
 INSERT ")" IN FRONT OF    .
         PRESS HELP FOR A DIFFERENT SUGGESTION.
```

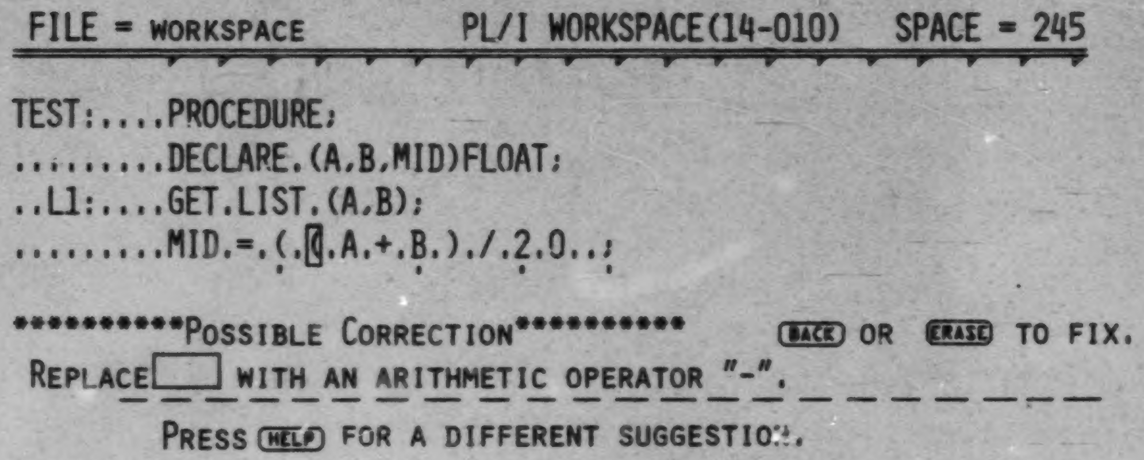
FIGURE 3. EDIT–TIME ERROR ANALYSIS

G)



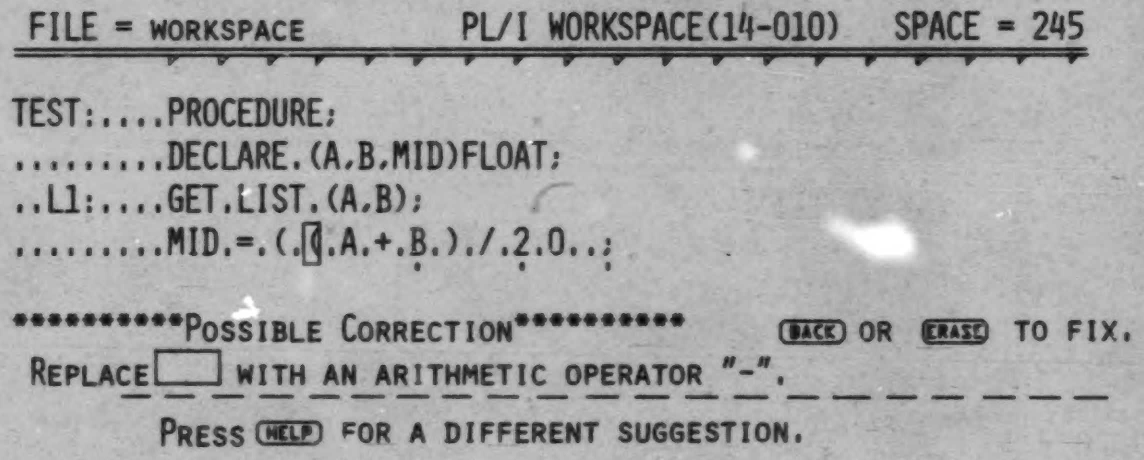
```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(A,B);
.........MID.=.(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********      BACK OR ERASE  TO FIX.
 INSERT ")" IN FRONT OF    .
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
         PRESS HELP FOR A DIFFERENT SUGGESTION.
```

H)



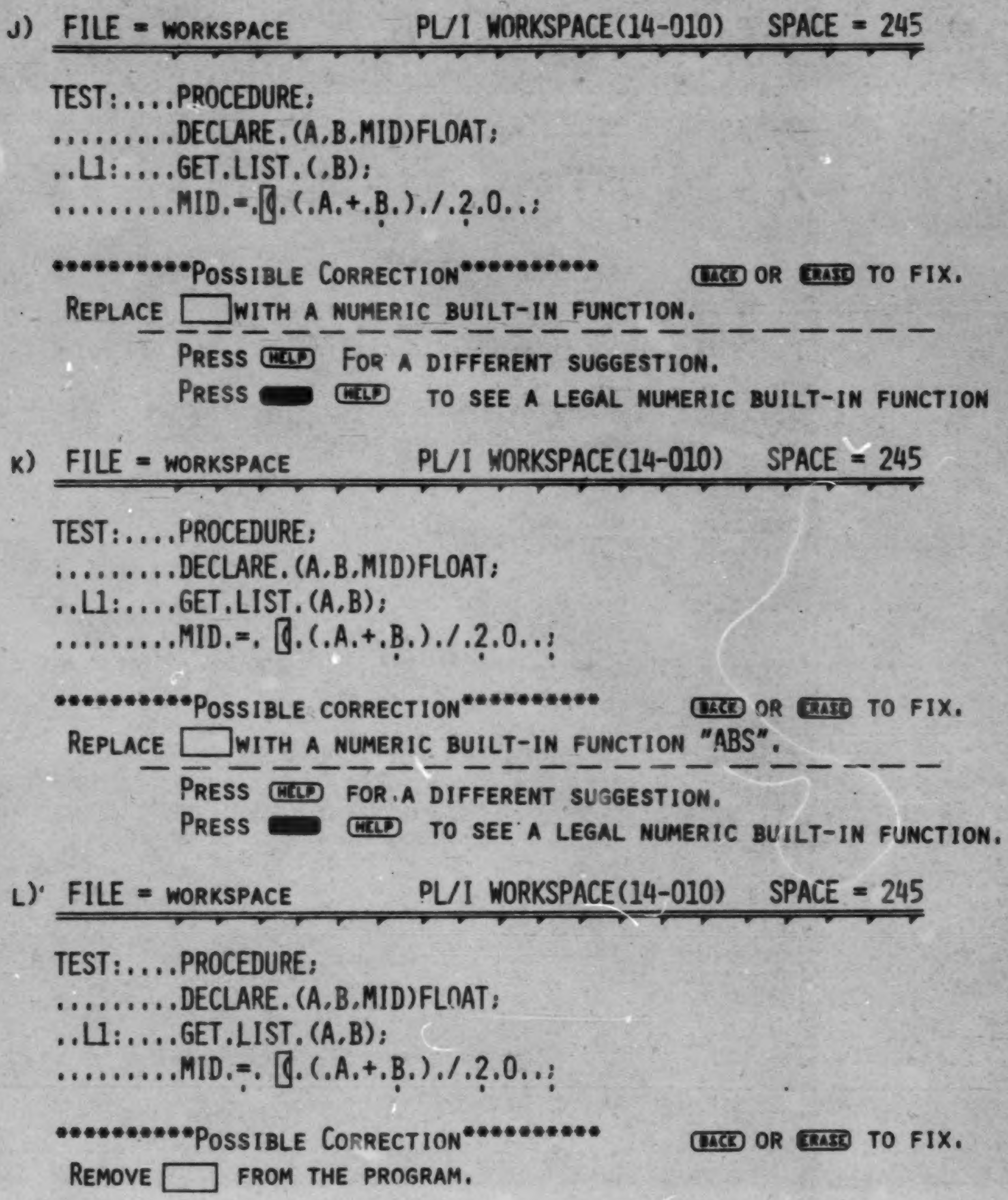
```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(A,B);
.........MID.=.(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********      BACK OR ERASE  TO FIX.
 REPLACE     WITH AN ARITHMETIC OPERATOR "-".
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
         PRESS HELP FOR A DIFFERENT SUGGESTION.
```

I)



```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(A,B);
.........MID.=.(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********      BACK OR ERASE  TO FIX.
 REPLACE     WITH AN ARITHMETIC OPERATOR "-".
     - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
         PRESS HELP FOR A DIFFERENT SUGGESTION.
```

FIGURE 3. EDIT-TIME ERROR ANALYSIS



J)
```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(,B);
.........MID.=.▯(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********        (BACK) OR (ERASE) TO FIX.
  REPLACE ▭ WITH A NUMERIC BUILT-IN FUNCTION.
      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        PRESS (HELP)  FOR A DIFFERENT SUGGESTION.
        PRESS (▬) (HELP)  TO SEE A LEGAL NUMERIC BUILT-IN FUNCTION
```

K)
```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(A,B);
.........MID.=. ▯(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********        (BACK) OR (ERASE) TO FIX.
  REPLACE ▭ WITH A NUMERIC BUILT-IN FUNCTION "ABS".
      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        PRESS (HELP)  FOR A DIFFERENT SUGGESTION.
        PRESS (▬) (HELP)  TO SEE A LEGAL NUMERIC BUILT-IN FUNCTION.
```

L)
```
FILE = WORKSPACE          PL/I WORKSPACE(14-010)   SPACE = 245

TEST:....PROCEDURE;
.........DECLARE.(A,B,MID)FLOAT;
..L1:....GET.LIST.(A,B);
.........MID.=. ▯(.(.A.+.B.)./.2.0..;

**********POSSIBLE CORRECTION**********        (BACK) OR (ERASE) TO FIX.
  REMOVE ▭  FROM THE PROGRAM.
```

FIGURE 3. EDIT—TIME ERROR ANALYSIS

point of modification. The symbols selected for insertion and replacement are those symbols that would be accepted by the parser at the point of modification. This information is immediately available from the parser's table.

The symbols that can be used as a modification are not only the terminals of the language, but include the non-terminals as well. (Non-terminals become procedures in the recursive descent parse.) This allows the CAPS error handler to communicate in general terms such as "expressions", "statements" and, as in Figure 3f, "built-in functions". The more conventional automatic schemes are restricted to a vocabulary of terminal symbols. The phrase used for each non-terminal is supplied by the language implementor. Note that as the recursive descent parser is backed up it returns to procedure levels closer and closer to the procedure level of the sentence symbol and thus the error handler first gives suggestions for local modifications and then, only when these fail, will it suggest more global modifications.

Whenever a non-terminal is involved in a modification, the student is given the opportunity to see what a non-terminal of that type looks like (Figures 3j & 3k). Thus whenever a general modification is suggested, more detailed suggestions are available if needed. Currently he is shown only the first symbol of a non-terminal - just to get him started - but it would be possible with considerable effort to display the entire non-terminal in some convenient form such as BNF.

3.2.4. Execution

Execution of the student's program is performed by interpreting the internal representation produced by the editor. The internal representation is just a tokenized form of the program, laid down in the same order as it appears on the screen, including spacing information and comments. This internal representation clearly favors the editor, but there is good reason for this. There is only enough storage for one copy of each student's program and whereas response time can deteriorate at run-time without doing harm, quick response is essential during editing. In a sense, expanding the execution time scale is actually beneficial in that it makes the student more aware of the relative efficiencies of different algorithms. Fortunately, the important symbols of the program are clearly marked and passed to an interpreter, so reparsing each statement every time it is executed is not an unduly burdensome task for the interpreters.

For each new language introduced into CAPS a new interpreter must be designed and implemented in TUTOR. Since all interpreters must work with the same form of internal representation and use the same symbol table structure, many modules can be borrowed from the existing interpreters. So the interpreters are not really built from scratch, but they are a long way from being table driven like the editor.

For the most part, the features provided by the interpreters depend on the programming language, but one thing they must all provide is a trace facility. Two levels of trace are provided: flow-of-control only, and flow-of-control with variable assignments. The flow trace outlines the keyword of each statement as it is executed. The variable trace displays the new value of a variable each time it is changed. Students are encouraged to run with flow trace enabled so that they can see their algorithm execute and obtain a better feeling for what each statement does. There is one danger, however: after watching their program execute with a flow trace, students often think something has gone wrong when it executes so quickly without trace - even though the output produced is identical!

3.2.5. Execution-time error handling

The main advantage of interactive error handling is that the programmer is on hand and can interact with the system to debug his program. In CAPS, with its emphasis on aiding instruction, the goal is to discuss with the student the nature of the error, obtain from him information about what specific sections of his program are supposed to do, and finally to suggest changes that he might make that would prevent future occurrences of the error. An additional goal is to perform these tasks in an easy-to-follow, orderly manner so that (1) the novice programmer does not become confused while using the system and (2) he eventually learns how to debug programs by himself.

In CAPS, the interactive debugging session is directed by the system and not by the student. This is essential because the beginning programmer does not know what questions to ask; he does not know how to debug. An added benefit of this is that the student does not have to learn to command language for the debugging package.

The CAPS run-time error analyzer [4] is given control when one of the interpreters has detected an obvious anomaly in the run-time environment - a zero divisor or a subscript out of range, for example. The task of the error analyzer is to help the student locate the cause of this anomaly.

In general an anomaly will be caused either by an error in assigning values to some of the variables involved or by an error in setting up storage for one of these variables. To locate the cause of the error in the first case, the analyzer must reverse-execute the program, searching back through the history of execution to find out where and how those variables were assigned the troublesome values. Similarly, in the second case, the analyzer locates the cause by reverse-executing the program to the point of instantiation of the variable and then searching back through the history of execution to find where and how the variables that controlled the instantiation were assigned the troublesome values.

In both cases the analyzer engages in a discussion with the student while internally it is reverse-executing his program from the point of error looking for assignments to one of the troublesome variables involved in the error. Once such an assignment has been found, it is shown to the student. If this does not help the student locate the cause of his error, the error analyzer then looks at the expression that computed the troublesome value to see if any suspicious conditions are present that could be the cause of the error. Whenever a possible cause is located an appropriate discussion about the condition is initiated.

For this analysis, a common misconception table is used. The table contains information about situations in the language that are potential trouble spots as well as templates for the actual discussions that should be initiated if the situation is recognized. Discussions located in the table are referenced during expression analysis by an (operator, value) pair in the following way: As an expression is being analyzed for the cause of the error, each operator and the value it returned are looked up in the table. If the pair is present, the discussion corresponding to it is initiated, otherwise analysis continues. The analysis of expressions is done by a pre-order traversal of the expression tree. This creates a top-down analysis of the expression so that if there is more than one common misconception in a given expression, the "outer-most" one is discussed first. Then, if the student needs more help, the inner ones are discussed to give the student more details about the situation. Note that there may be more than one (operator, value) pair corresponding to any one common misconception, and more than one misconception may be associated with a pair.

If no common misconceptions are present or none of the discussions reveal the cause of the error to the student, he is asked to identify the variables in the expression that do not seem right to him. The variable changed by the expression just analyzed is deleted from the list of troublesome variables and those variables selected by the student are added. Reverse-execution is then resumed looking for changes to something on the list.

Note that the selection of variables by the student is not necessary, but it does permit oracular pruning of the search tree and makes it clearer to the student how the analyzer is proceeding.

Error analysis terminates when one of the following situations arise: (1) the last troublesome variable on the list is assigned a constant (The constant is incorrect.) or appears in an input statement (The value input is incorrect.), (2) the student chooses to edit his program (The common misconception most recently presented is probably the cause.) or (3) the student does not select replacements for the last troublesome variable (The student has not understood his problem well enough to know when the variables he is using have improper values.) Assistance in these cases is beyond the scope of the error analyzer.

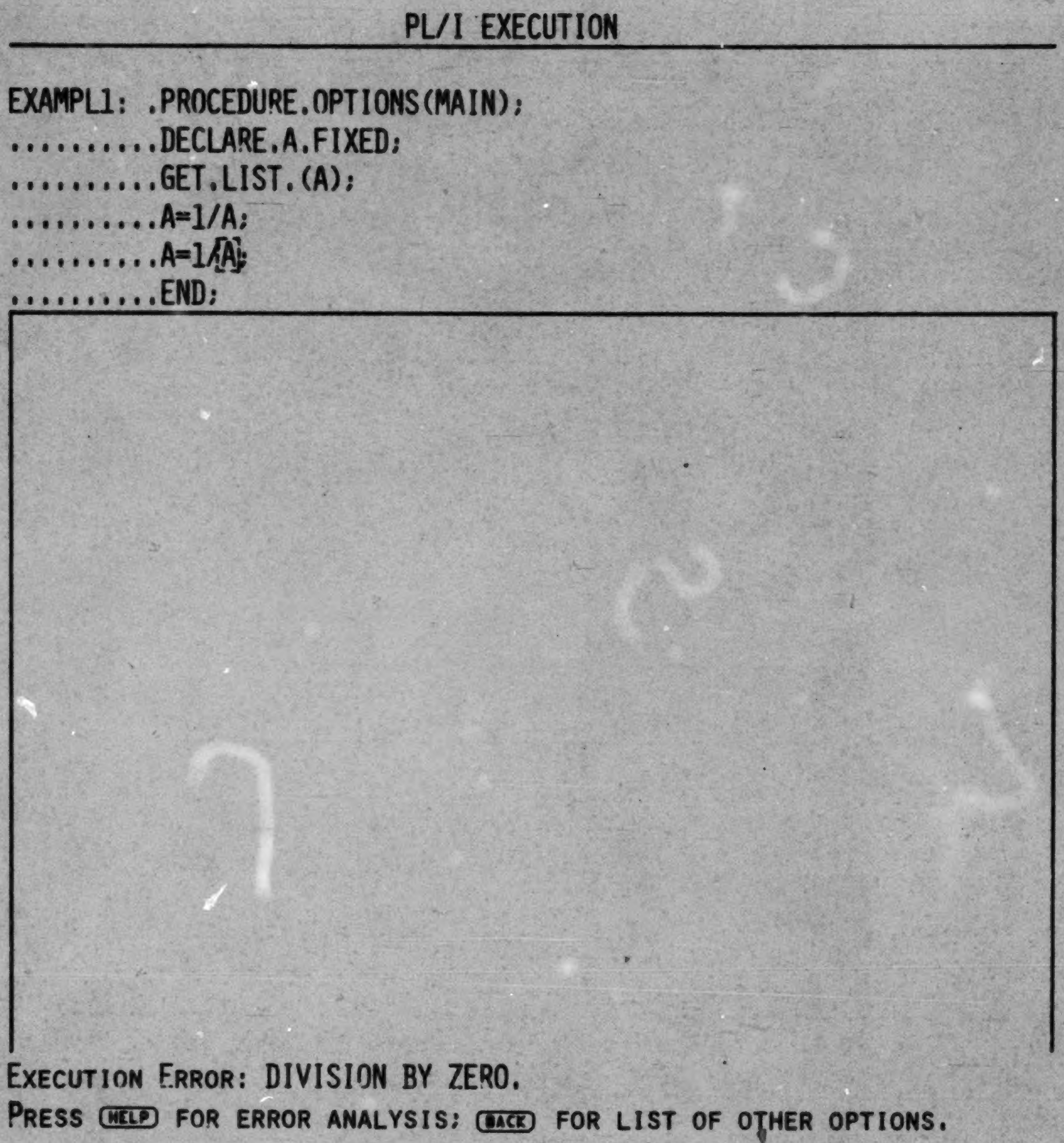
The example in Figure 4 should demonstrate the features of the run-time error analysis. Assuming the student has prepared the

PL/1 program shown, he would receive the execution error as indicated at the bottom of Figure 4a. If he requests help, he receives the display in Figure 4b showing him where the variable A was assigned the troublesome value 0. When the student requests more help, the expression computing the value assigned to A is analyzed. In this case (fixed divide, 0) appears in the common misconception table for PL/1 and so the information shown in Figure 4c is presented to the student. If the student requests more help, the pre-order traversal continues. As this is done, the programmer is asked about the reasonableness of values of relevant variables (Figure 4d). If the student asserts that it is reasonable for A to have the value 8, the analyzer must respond with the conclusion shown in Figure 4e. Since there are no more troublesome variables to search for, further requests for help are answered with the message shown at the bottom of Figure 4f. If the student admits that 8 is an unreasonable value for A, the statement that gave A that value is located and shown to the student (Figure 4g). Since this is an input statement involving the only troublesome value, the analyzer terminates with the conclusion shown in Figure 4h.

The run-time error analysis algorithm is essentially language independent. The language dependent information is contained in the common misconception table. In addition, each language interpreter must generate a history of execution in a standard form (similar to that used by Zelkowitz [15]) and provide a routine to construct expression trees from the internal representation of a statement.

3.3. Evaluation

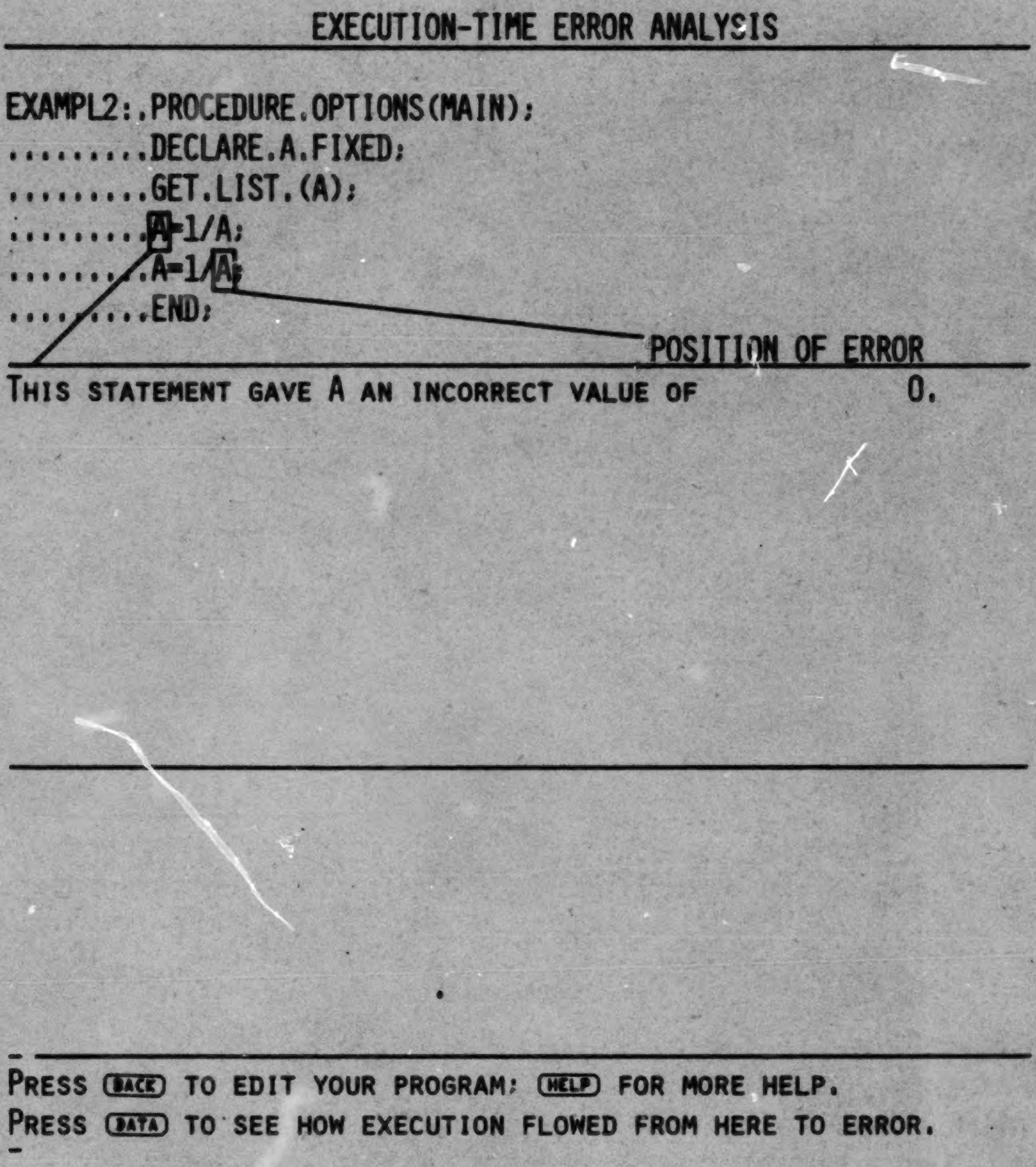
CAPS is an experiment in extending the concept of pedagogic programming systems to the interactive environment. The logical setting



```
                              PL/I EXECUTION
EXAMPL1: .PROCEDURE.OPTIONS(MAIN);
..........DECLARE.A.FIXED;
..........GET.LIST.(A);
..........A=1/A;
..........A=1/A;
..........END;

EXECUTION ERROR: DIVISION BY ZERO.
PRESS HELP FOR ERROR ANALYSIS; BACK FOR LIST OF OTHER OPTIONS.
```

A) DETECTION OF ERROR. (STUDENT PRESSES HELP KEY.)

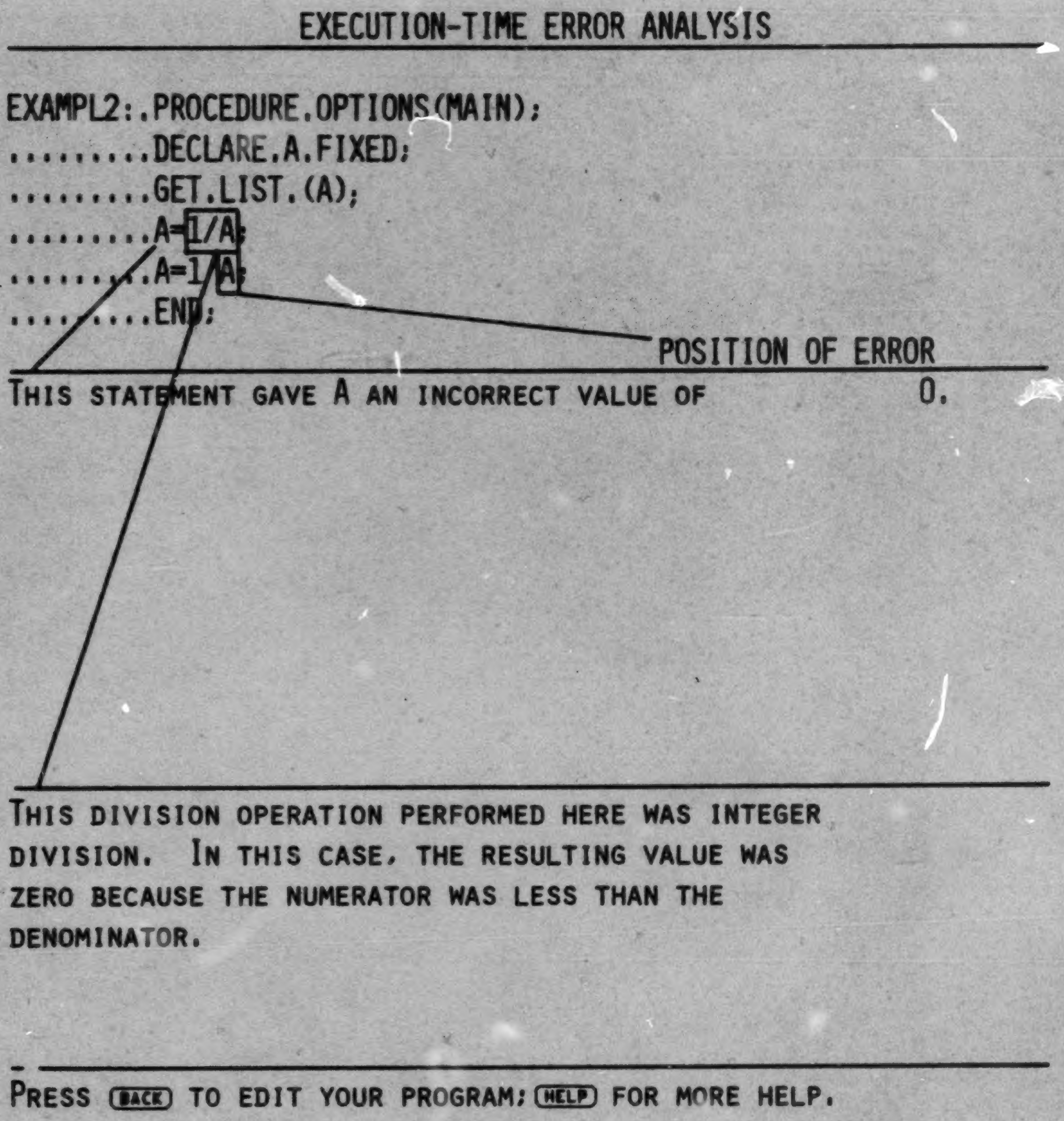
FIGURE 4. EXECUTION-TIME ERROR ANALYSIS



```
                        EXECUTION-TIME ERROR ANALYSIS
EXAMPL2:.PROCEDURE.OPTIONS(MAIN);
.........DECLARE.A.FIXED;
.........GET.LIST.(A);
.........A=1/A;
.........A=1/A;
.........END;
                                                 POSITION OF ERROR
THIS STATEMENT GAVE A AN INCORRECT VALUE OF                  0.




PRESS BACK TO EDIT YOUR PROGRAM; HELP FOR MORE HELP.
PRESS DATA TO SEE HOW EXECUTION FLOWED FROM HERE TO ERROR.
```

B) ASSIGNMENT THAT LEAD TO ERROR. (PRESSES HELP AGAIN)

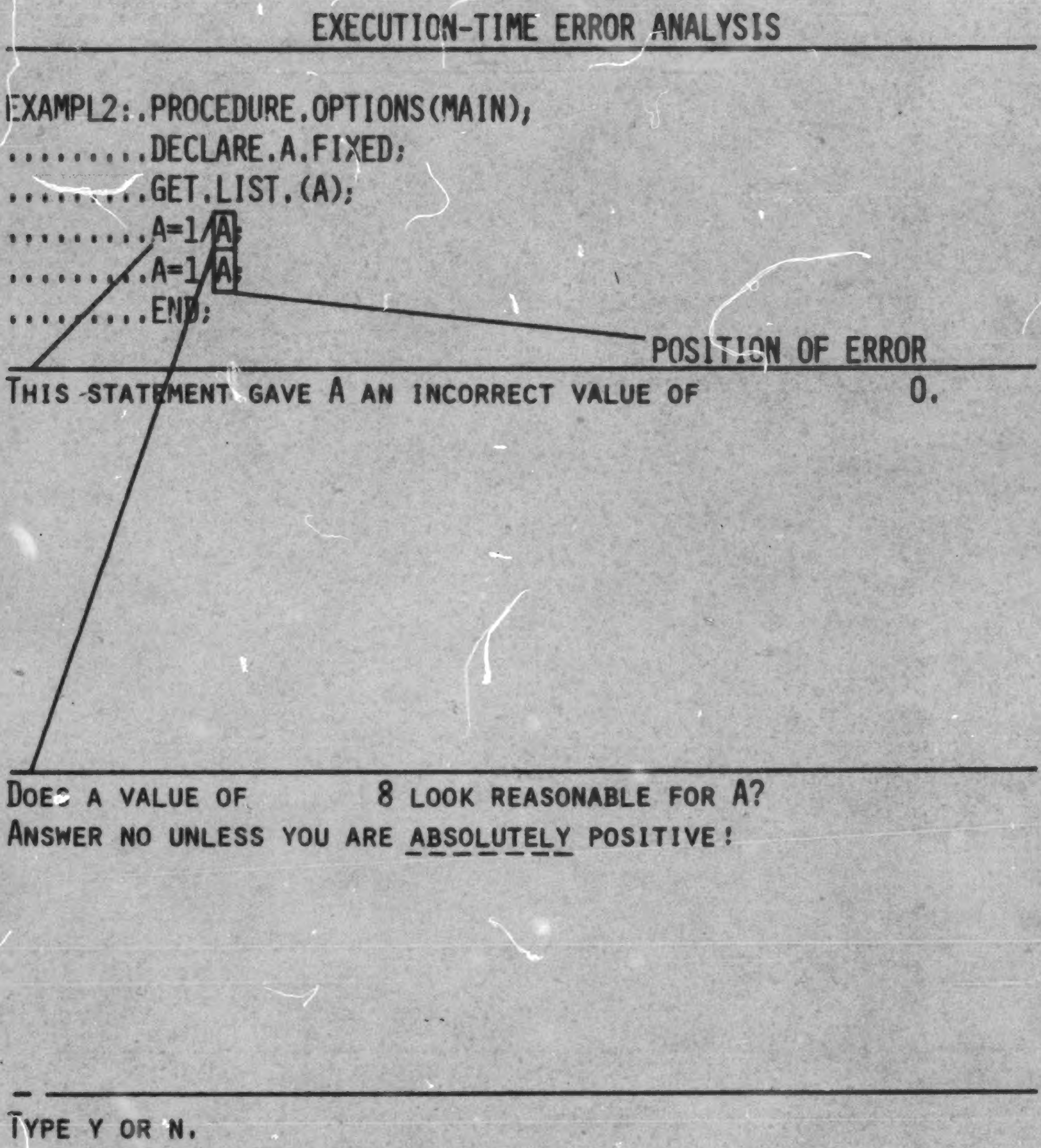
FIGURE 4. EXECUTION-TIME ERROR ANALYSIS



```
              EXECUTION-TIME ERROR ANALYSIS

EXAMPL2:.PROCEDURE.OPTIONS(MAIN);
.........DECLARE.A.FIXED;
.........GET.LIST.(A);
.........A=1/A;
.........A=1/A;
.........END;
                                    POSITION OF ERROR
THIS STATEMENT GAVE A AN INCORRECT VALUE OF           0.



THIS DIVISION OPERATION PERFORMED HERE WAS INTEGER
DIVISION.  IN THIS CASE, THE RESULTING VALUE WAS
ZERO BECAUSE THE NUMERATOR WAS LESS THAN THE
DENOMINATOR.


PRESS (BACK) TO EDIT YOUR PROGRAM; (HELP) FOR MORE HELP.
```

c) Possible cause of unexpected result. (Presses HELP again)

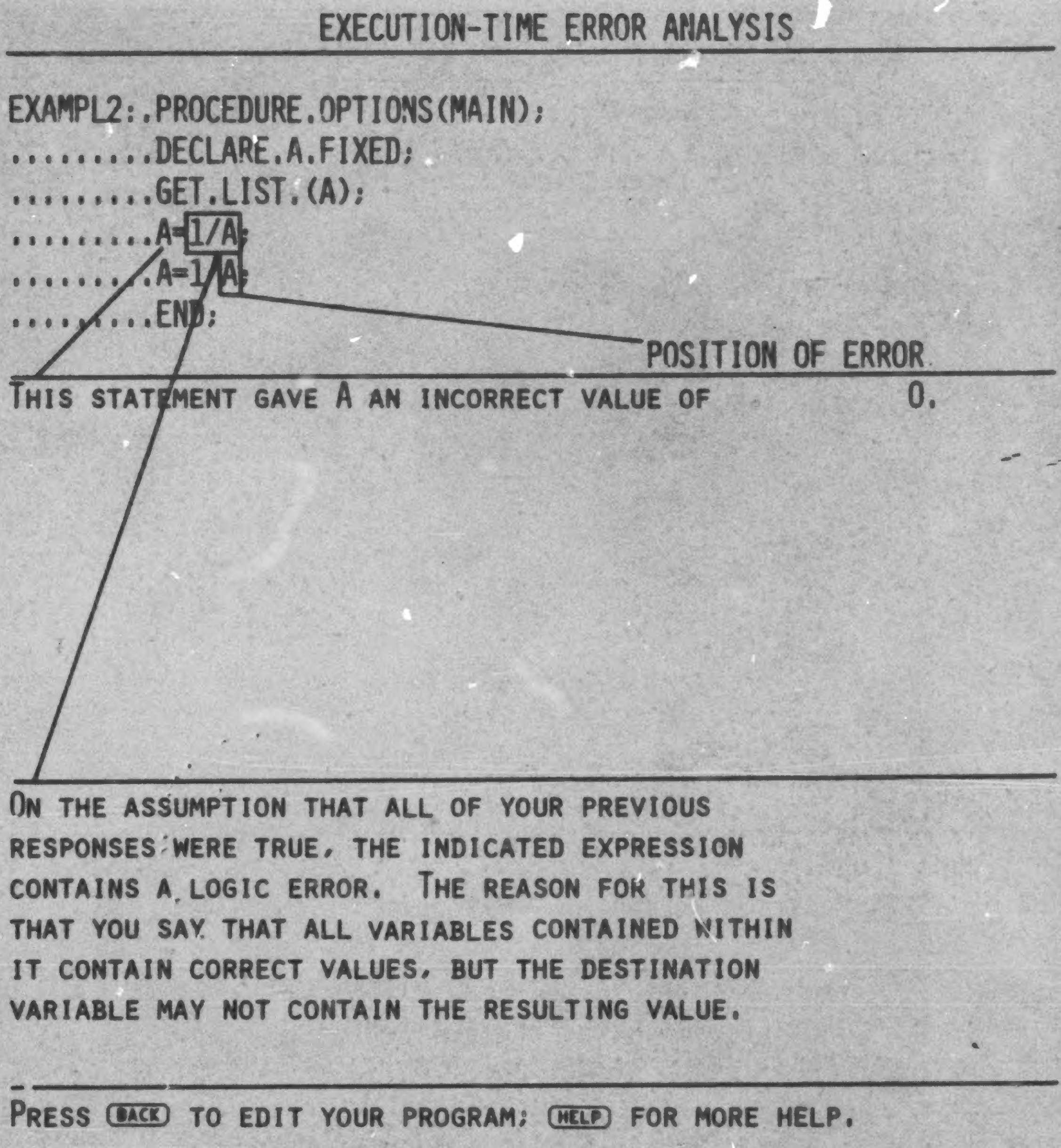
FIGURE 4. EXECUTION-TIME ERROR ANALYSIS



```
EXECUTION-TIME ERROR ANALYSIS

EXAMPL2:.PROCEDURE.OPTIONS(MAIN);
.........DECLARE.A.FIXED;
.........GET.LIST.(A);
.........A=1/A;
.........A=1/A;
.........END;
                                        POSITION OF ERROR
THIS STATEMENT GAVE A AN INCORRECT VALUE OF             0.

DOES A VALUE OF         8 LOOK REASONABLE FOR A?
ANSWER NO UNLESS YOU ARE ABSOLUTELY POSITIVE!

TYPE Y OR N.
```

D) Request for input from student. (Presses "Y" or "N")

FIGURE 4. EXECUTION-TIME ERROR ANALYSIS



EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2:.PROCEDURE.OPTIONS(MAIN);
.........DECLARE.A.FIXED;
.........GET.LIST.(A);
.........A=1/A;
.........A=1/A;
.........END;
```

POSITION OF ERROR

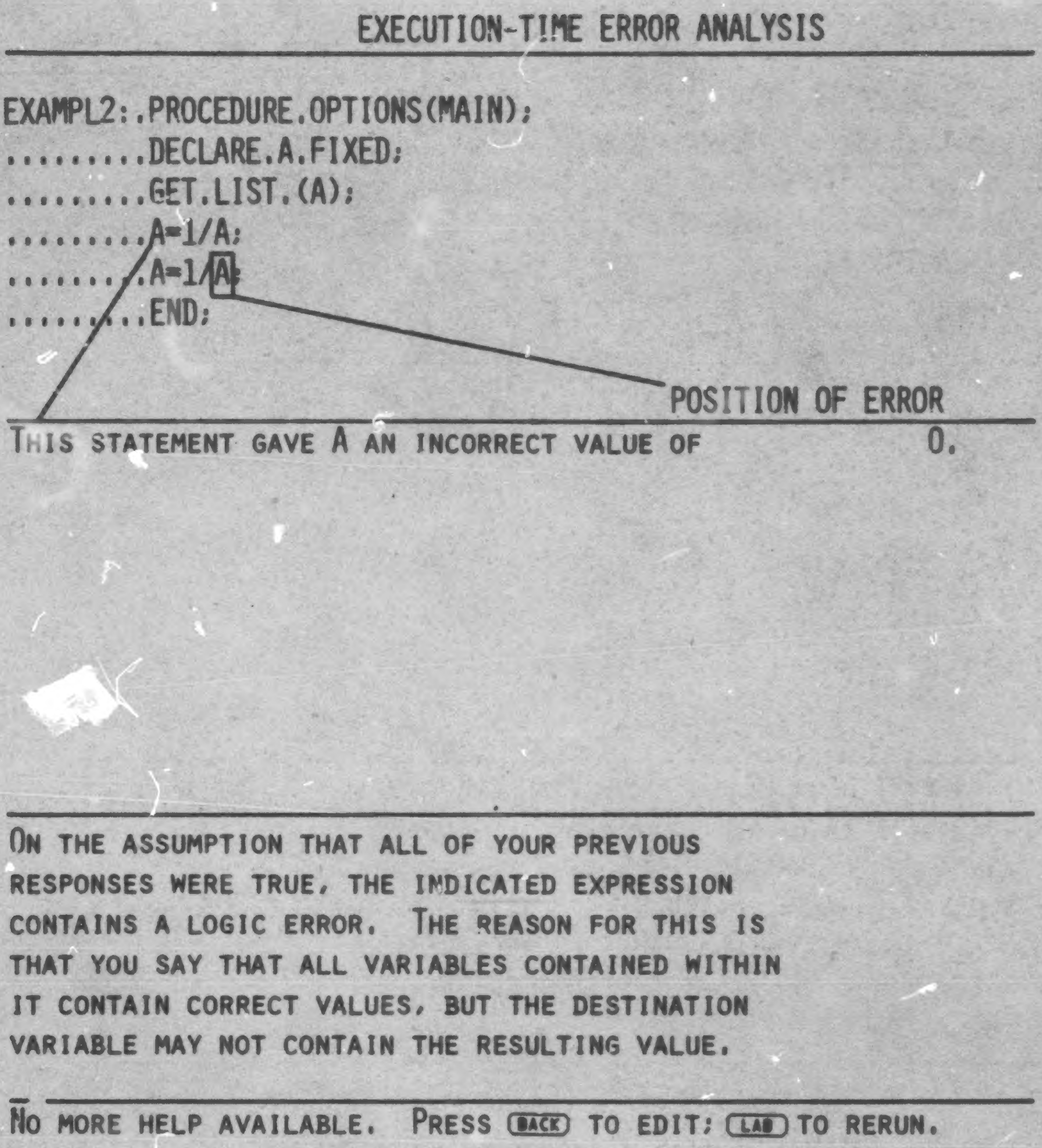
THIS STATEMENT GAVE A AN INCORRECT VALUE OF 0.

ON THE ASSUMPTION THAT ALL OF YOUR PREVIOUS RESPONSES WERE TRUE, THE INDICATED EXPRESSION CONTAINS A LOGIC ERROR. THE REASON FOR THIS IS THAT YOU SAY THAT ALL VARIABLES CONTAINED WITHIN IT CONTAIN CORRECT VALUES, BUT THE DESTINATION VARIABLE MAY NOT CONTAIN THE RESULTING VALUE.

PRESS BACK TO EDIT YOUR PROGRAM; HELP FOR MORE HELP.

E) DISPLAY GIVEN WHEN STUDENT TYPES "Y". (PRESSES HELP AGAIN.)

FIGURE 4. EXECUTION-TIME ERROR ANALYSIS



EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2:.PROCEDURE.OPTIONS(MAIN);
.........DECLARE.A.FIXED;
.........GET.LIST.(A);
.........A=1/A;
.........A=1/A;
.........END;
```

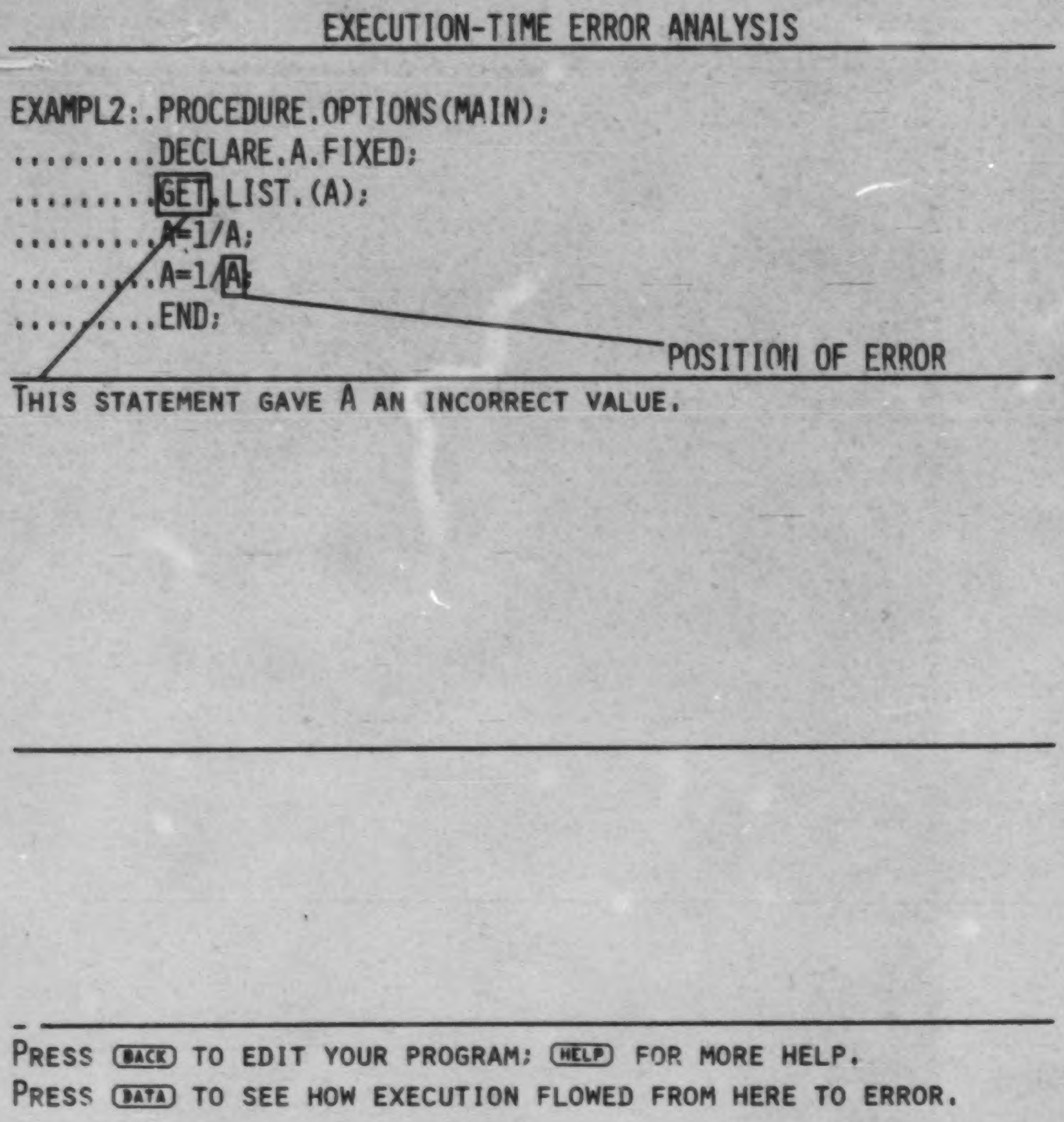
THIS STATEMENT GAVE A AN INCORRECT VALUE OF 0.

ON THE ASSUMPTION THAT ALL OF YOUR PREVIOUS RESPONSES WERE TRUE, THE INDICATED EXPRESSION CONTAINS A LOGIC ERROR. THE REASON FOR THIS IS THAT YOU SAY THAT ALL VARIABLES CONTAINED WITHIN IT CONTAIN CORRECT VALUES, BUT THE DESTINATION VARIABLE MAY NOT CONTAIN THE RESULTING VALUE.

NO MORE HELP AVAILABLE. PRESS BACK TO EDIT; LAB TO RERUN.

F) ALL HELP EXHAUSTED. (HELP KEY NO LONGER ACTIVE.)

FIGURE 4. EXECUTION-TIME ERROR ANALYSIS



EXECUTION-TIME ERROR ANALYSIS

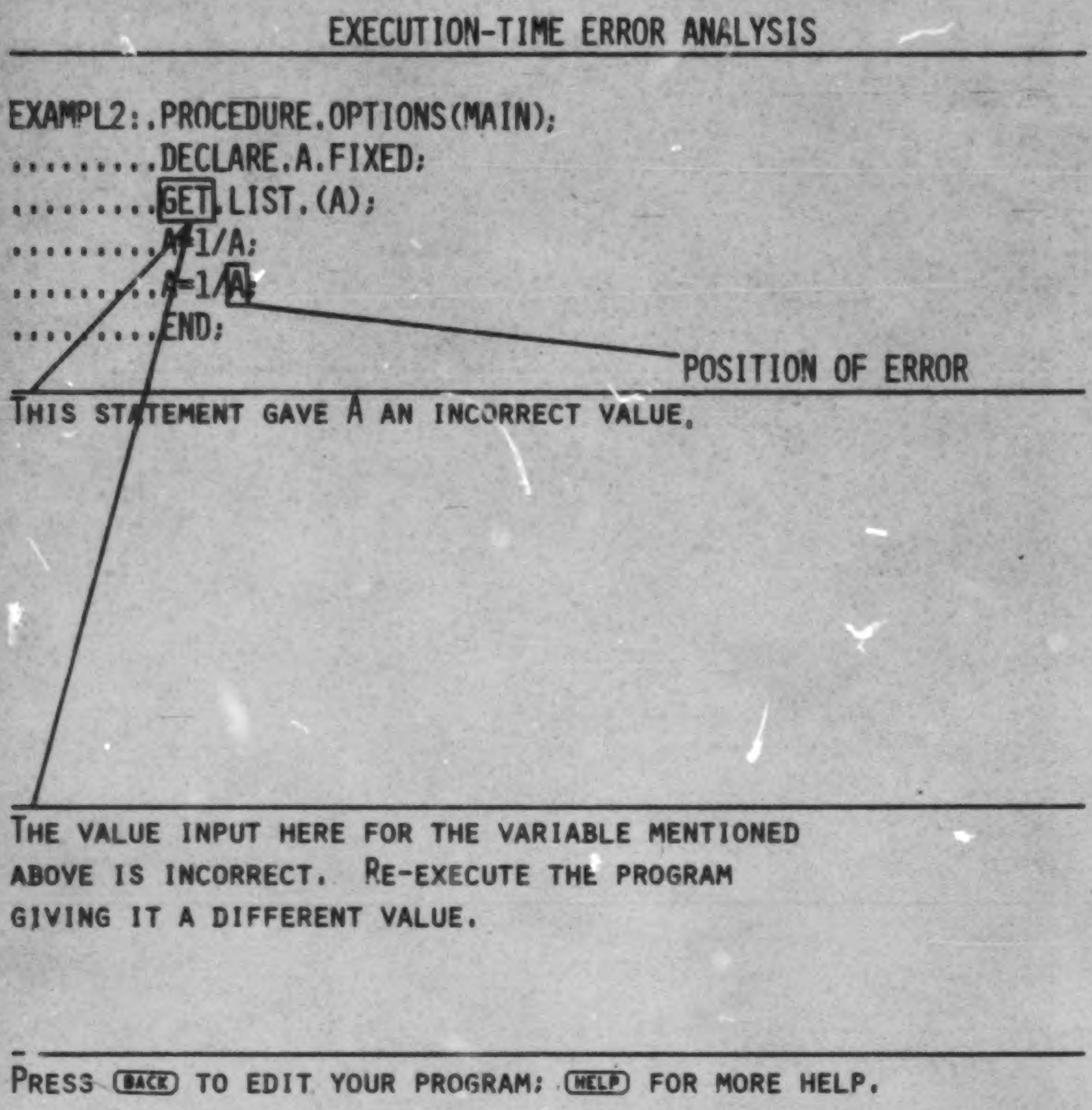
```
EXAMPL2:.PROCEDURE.OPTIONS(MAIN);
.........DECLARE.A.FIXED;
.........GET.LIST.(A);
.........A=1/A;
.........A=1/A;
.........END;
```

THIS STATEMENT GAVE A AN INCORRECT VALUE.

PRESS BACK TO EDIT YOUR PROGRAM; HELP FOR MORE HELP.
PRESS DATA TO SEE HOW EXECUTION FLOWED FROM HERE TO ERROR.

G) DISPLAY GIVEN WHEN STUDENT TYPES "N". (PRESSES HELP AGAIN)

FIGURE 4. EXECUTION-TIME ERROR ANALYSIS



EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2:.PROCEDURE.OPTIONS(MAIN);
.........DECLARE.A.FIXED;
.........GET.LIST.(A);
.........A=1/A;
.........A=1/A;
.........END;
```

POSITION OF ERROR

THIS STATEMENT GAVE A AN INCORRECT VALUE.

THE VALUE INPUT HERE FOR THE VARIABLE MENTIONED ABOVE IS INCORRECT. RE-EXECUTE THE PROGRAM GIVING IT A DIFFERENT VALUE.

PRESS (BACK) TO EDIT YOUR PROGRAM; (HELP) FOR MORE HELP.

H) CONCLUSION OF ERROR ANALYSIS

FIGURE 4. EXECUTION-TIME ERROR ANALYSIS

for such a system is a CAI facility - particularly one trying to teach computer programming. The facilities provided by PLATO IV seem ideal: impressive terminal hardware and software support; extensive built-in procedures for interactive dialogues, response analysis and graphics work; a convenient, yet basic time sharing facility; and a language that is sophisticated enough to make writing large programs manageable, but close enough to the machine and the system to make efficient use of their limited facilities.

The experiment has been successful only with qualification. So far, over 500 students have used CAPS while learning PL/1 and FORTRAN. The diagnostic assistance in the interactive environment is clearly superior to any batch system or interactive system for this class of users. Unfortunately CAI systems are not yet up to the task of supporting the degree of computing and interaction needed by a system such as CAPS. When PLATO IV is handling 500 students simultaneously, even if only 30 terminals run CAPS, the student gets annoyingly slow response - slow, even for the "hunt-and-peck" typist writing in an unfamiliar language. The average CAI lesson uses about 2ms of CPU time per second (about half of this is system overhead): CAPS requires five to ten times this CPU rate to give satisfactory response.

3.3.1. Possible improvements

Four suggestions for improving the CPU time requirement of the editor have been made. The first involves minor recoding of critical sections of the compiler and would give a minor improvement in speed. Two such changes are to 1) stop displaying the "space left" indicator on the student's screen -- 2.5% improvement, and 2) stop doing unnecessary

-stoload- commands--2% speedup. (The compiler is reexecuting the same -stoload- command once for every token, which is unnecessary, at least for the PL/1 compiler.)

The second suggestion involves recoding the lexical analyzer or parser in TUTOR, rather than having TUTOR code interpret these tables. This would give an unknown amount of speedup, estimated at 20% for the lexical analyzer, more for the parser, but reduces the generality of the driver program.

The third method proposed for speeding up the editor is to process one line of source text at a time rather than one character at a time. This moves the collection process from the TUTOR lesson to the PLATO system, and allows the user to correct errors in his current line before giving it to the lexical analyzer and parser. This method has been implemented and tested, and does reduce the CPU time used, but the system is then unable to respond immediately to an invalid character or token.

The fourth method suggested is to move semantic checking and symbol table construction to a later pass. Timing tests conducted by L. White suggest anticipated improvement is probably more than 20%.

Of the four possible improvements, this last is least objectionable since it retains the table-driven nature of the CAPS editor and still permits instantaneous analysis of many errors commonly made by the programming neophyte. In addition, the second compiler pass permits generation of a representation of the program that is more easily interpreted so that execution speed is improved as well.

3.3.2. Work in progress

S. Nakamura [8] has implemented an experimental FORTRAN compiler using the two pass organization with encouraging results. His compiler is diagrammed in Figure 5.

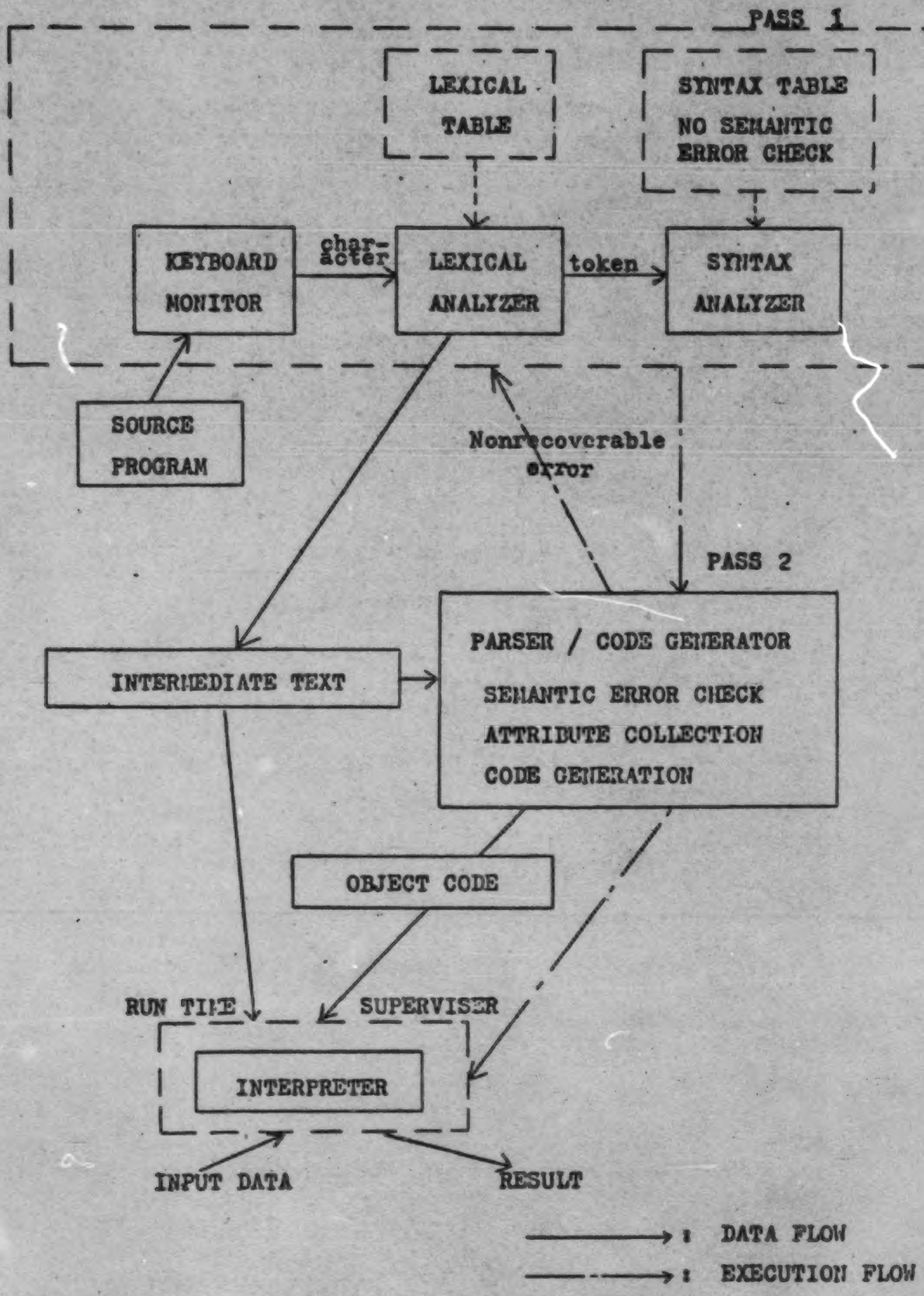


Figure 5: Reorganized Compiler (Nakamura)

Paralleling Nakamura's experimental compiler development is an effort by T. Fishman to improve the parser module of the editor. The current parsing tables must be programmed by the compiler-writer. Considerable knowledge and effort is required to implement a language and the process is prone to error. Since, in the reorganized editor, the parser is responsible only for syntactic details, more formal, grammar-driven parsing methods are possible. In particular, Fishman is investigating the possibility of using LALR parsing techniques in CAPS. The advantages of an LALR parser is that both the parser and the parsing table will be more compact than the current parser and the table can be constructed directly from the formal grammar specifying the programming language.

The output of the new parser will have more of the syntactic structure of the program built into the intermediate text, which will permit the editor to deal directly with syntactic units such as statements and expressions. As long as these syntactic units are left intact during editing, the parser will not have to be called to unparse them as the cursor moves past.

The structured intermediate text should also permit the second pass to be table-driven to a greater extent that it is now. B. Speelpenning is currently revising Nakamura's FORTRAN compiler to use the output of Fishman's LALR parser to make use of this structure.

3.4. Conclusion

The attempts by Nakamura, Fishman, and Speelpenning are essential to the continued existence of CAPS at its current level of diagnostic assistance. Experience has shown that in its original formulation, CAPS would be intolerably slow in rigorous classroom use and this degraded performance is directly a consequence of providing enhanced diagnostic assistance. So should the current research prove fruitless, we must conclude that at the level of CPU processing currently available, the desired level of automatic error diagnosis is unattainable on PLATO.

References

[1] Alpert, D. and Bitzer, D. L. Advances in computer based education. *Science*, 167, (20 March 1970), 1582-1590.

[2] Conway, M. E. Design of a separable transition diagram compiler. *CACM*, 6, 7, (July 1963), 396-409.

[3] Conway, R. W. and Wilcox, T. R. Design and implementation of a diagnostic compiler for PL/1. *CACM*, 16, 3, (March 1973), 169-179.

[4] Davis, A. M. An interactive analysis system for execution-time errors. (Ph.D. Thesis), Technical Report No. UIUCDCS-R-75-695, University of Illinois at Urbana-Champaign, Department of Computer Science, January 1975.

[5] Holt, R. C. and Wortman, D. B. Structured subsets of the PL/1 language. CSRG-27, Computer Systems Research Group, University of Toronto, October 1973.

[6] Levy, J. P. Automatic correction of syntax errors in programming languages. (Ph.D. Thesis), Technical Report No. TR71-116, Cornell University, Computer Science Department, December 1971.

[7] Lomet, D. B. A formalization of transition diagram systems. *Journal of the ACM*, 20, 2, (April 1973), 235-257.

[8] Nakamura, S. Reorganization of an interactive compiler. (M. S. Thesis), to appear as Technical Report, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1976.

[9] Nievergelt, J., Reingold, E. M. and Wilcox, T. A. The automation of introductory computer science courses, Gunther, A., et al. (Ed.), International Computing Symposium 1973, North Holland Publishing Co., 1974.

[10] Sherwood, B. A. The TUTOR language. Computer-Based Education Research Lab., University of Illinois at Urbana-Champaign, 1974.

[11] Tindall, M. H. An interactive table driven parser. (M. S. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, August 1975.

[12] Tindall, M. H. Table driven compiler error analysis. (Ph.D. Thesis), University of Illinois at Urbana-Champaign, Department of Computer Science, October 1975.

[13] Tixier, V. Recursive functions of regular expressions in language analysis. CS58, Computer Science Department, Stanford University, March 1967.

[14] Wilcox, T. R. An interactive table driven diagnostic editor for high level programming languages. Department of Computer Science, University of Illinois at Urbana-Champaign, 1975.

[15] Zelkowitz, M. Reversible execution as a diagnostic tool. (Ph.D. Thesis), Cornell University, Computer Science Department, January 1971.

4. The GUIDE information and advising system (D. R. Eland, J. L. Pradels)

4.1. Introduction

This section discusses the GUIDE - an information system which was developed to advise students in the use of the ACSES library of PLATO lessons (Eland [1]). In an abstract sense, the information managed by the GUIDE is a set of objects, a set of relations between those objects, a set of attributes possessed by those objects, and a set of relations between those attributes. In concrete terms, the objects described above are lessons; the relations between the lessons are relationships like prerequisite, sequel, etc.; the attributes of the lessons are concepts in the discipline of computer science; and the relations between the concepts are relationships like generic, synonym, etc. Conceptually, this collection of information is viewed as an abstract "information space" which a student can explore when searching for lessons which deal with a topic he is interested in studying. Actually, it is useful to view this information space as a sort of "dual space": the lessons and relations between lessons form a "lesson space", the concepts and relations between concepts form a "concept space", and the two spaces are interconnected through keywords (i.e., concepts which have been explicitly attached to a lesson as a description of the content of that lesson).

By typing requests in the English language, a student can ask questions about the library of lessons, the lessons and performance required for a particular course, past performance in lessons previously taken, or requests for lessons on the various topics in computer science. The GUIDE will then respond with the desired information, explain why the request cannot be accepted, or ask for some feedback to help clarify the intent of the request. In addition, the student can initiate a graphics mode of interaction with the GUIDE which enables him to freely explore the information which is maintained by the system.

The GUIDE also functions as an administrative tool in addition to assisting students in finding lessons of interest. The information structuring and retrieving capabilities of the GUIDE can be used effectively in managing any relatively large body of information which possesses a rich structure of interconnections. In fact, the GUIDE has been adapted by the PLATO foreign languages group to manage their collection of lessons.

The essential components of the GUIDE are shown in the block diagram in Figure 1. The blocks enclosed by the dotted line represent the various areas of the database, those outside the dotted line represent the different processing algorithms of the system, and the solid lines and arrows represent the flow of information. In addition to generating the final response, the processing algorithms will also provide appropriate feedback messages when additional information is needed to process a request. Let's discuss the various components in more detail.

4.2. The GUIDE database

In the early design stages of the GUIDE, several distinct files of information were envisioned for the system, such as a lesson catalog, course outlines, a classification schedule, and the translator's vocabulary. However, the success of the final design was dependent to a large extent on the integration of all this data into a single database.

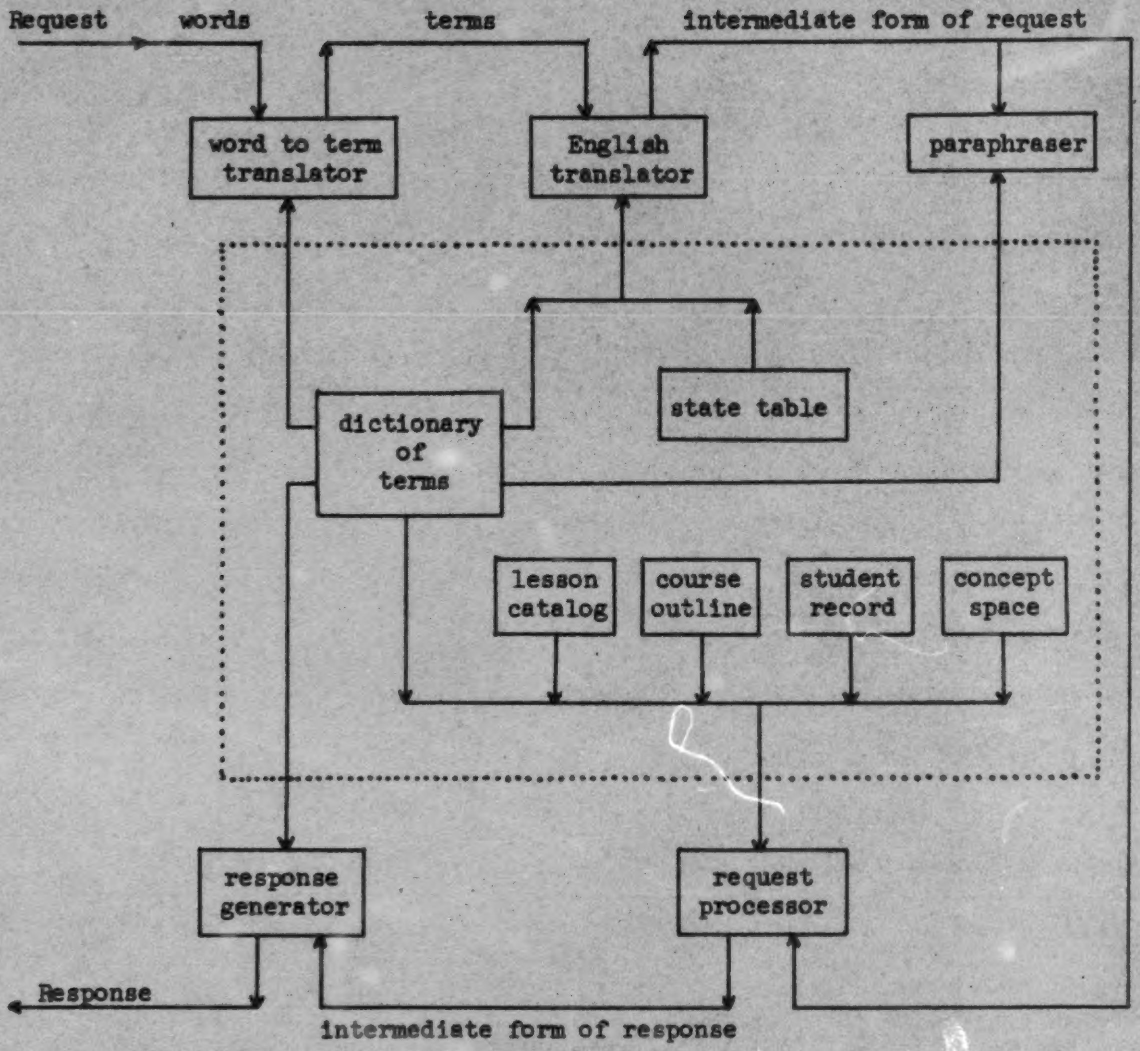


Figure 1: Block Diagram of the GUIDE

The performance of the GUIDE would suffer appreciably if the system were to lose the savings in memory space and processing time which have been achieved by utilizing an integrated database.

4.2.1. The dictionary of terms

The primary mechanism by which the database has been integrated is through the introduction of the concept of "terms". Any word or phrase known to the system (e.g. the lesson name "fortif" or the grammatical phrase "do you have" or the concept "do loops") is considered to be a term. All information known about a term is stored physically adjacent in memory. (Thus, knowing the name of a term makes all information about that term immediately available.) This packet of information -- a term and everything known about it -- forms a logical "record". The different types of terms (lesson names, concepts, course names, author names, and grammatical words) each have a different record format since the detailed information relevant to each is different. All records are of variable length so that optimum space utilization can be achieved.

Terms are located by hashing; each term is associated with a unique number which can be viewed as an accession number, and which remains invariant as long as the term is in the database. All interconnections between terms are made via this term number, which means records can be freely moved for memory mangagement with no need to make any changes at the record level -- only the record directory needs updating when this is done.

The result of using this "term concept" is that the entire GUIDE database (except for the state table and student record) is simply a sequence of term records. There is no physical entity which forms the lesson catalog, course outlines, or concept space. Rather, these are logical entities defined by the set of lesson records, course records, and concept records respectively.

4.2.2. State table

The state table is utilized by the English translator as it transforms the original request into an intermediate representation for further processing. Each grammatical word in the database is associated with one or more "input classes". Whereas lesson names, concepts, course names, and author names have an input class of 1, 2, 3, or 4, respectively, regardless of the current state of the English translator, the input category of grammatical words is a function of the current state of the translator. This mechanism was introduced to handle the situation of words which have different meanings in different contexts.

Corresponding to each possible term input class and each state of the finite state automaton which drives the translator is a table entry which determines the next state to be entered by the automaton and the output symbol to be generated (the output symbol becomes a part of the intermediate representation of the original input request). There is also a special level of confidence flag used to mark certain situations for special processing.

4.2.3. Lesson catalog

The lesson catalog is simply the set of all lesson records. Each record contains the following fields: lesson name, abstract, descriptive keywords, relationships to other lessons (prerequisite-sequel, exercise for-background for, etc.), authors, lesson type (instructional, exam, etc.), type of exercises (drill, simulation, etc.), level of difficulty, expected completion time, expected average grade, lesson development status, and several fields for interfacing with PLATO system routines.

4.2.4. Course outline

Each course outline record contains the course name, a list of concepts from the concept space (section 4.2.6) relevant to the course, and a list of lessons and the date by which each lesson should be completed. This latter list essentially imposes a prerequisite-sequel relationship on the lessons used within each course.

4.2.5. Student records

Because of the overwhelming amount of data which had to be maintained, student records are not actually a part of the GUIDE database. Rather, we use the storage space which the PLATO system automatically allocates to each student registered on the system. Such a record contains the student's name, course, and a list of each lesson entered by the student (with information on number of times each lesson was entered, elapsed time spent in each lesson, etc.).

4.2.6. Concept space

Every information retrieval system is faced with the problem of bridging the gap between a user's perception of the subject matter and the actual information which is stored in the system. The concept space was introduced into the design of the GUIDE to help bridge that gap.

While no one has yet developed a universally accepted technique for organizing a body of knowledge, there is some consensus that a useful point of view is to model knowledge of a subject as a network built of concepts and relations. Hence the data structure for the GUIDE concept space is simply an abstract graph where the nodes of the graph are concepts and the arcs are relations between concepts. The choice of this extremely simple yet powerful model was fully vindicated when it was put to use. It was found to be adequate to incorporate the synonym dictionary, the hierarchical classification scheme, and the term clusters which were originally proposed as separate components of the concept space. Also, it serves as a keyword index (holding all keywords which have been attached to lessons) and a thesaurus (holding all subject-matter terms known to the system and showing how they are related). Furthermore, when it was desired to build an index to the library of lessons, the concept space already provided the necessary mechanism to do so. And finally, it was possible to specify the structure of a course by means of concepts rather than a listing of lessons, again simply by utilizing the mechanisms available in the structure of the concept space.

It should be emphasized that the word "concept" is used rather loosely. Any word or phrase which is the name of a node in the concept space is called a "concept". Similarly, the relations used in the concept space were chosen so as to be intuitively clear to the user. These relations could be readily extended if it proved desirable to do so, or if a universally accepted set of relations among topics were to evolve.

The relations currently in use are: generic-specific, containor-containee (which imposes an index structure on the lesson library), related (for concepts related, but not by one of the more specific categories), synonym, owner-member, and prerequisite-sequel.

A concept record, then, consists of the concept name, a list of lessons in which the concept is a keyword, and a list of relationships with other concepts.

4.3. Word to term translator

The word to term translator accomplishes the task of transforming a sequence of input words into a sequence of terms recognized by the system. This is done by extracting the leftmost word of an input request, applying a hash function to that word, and looking in the hash table in the appropriate location. Entries are arranged in the hash tabe in such a way that it is possible to extract from the original request the longest possible substring which matches a term known to the system. The progress of the translation process is communicated to the student by underlining each term when it is found in the term dictionary.

4.4. English translator

The English translator accomplishes the task of transforming a sequence of input terms into an intermediate representation for further processing. The translation is not based on an elaborate linguistic analysis; rather, the translator searches through a space of partial meanings determined by an analysis of the domain of discourse. The translator's approach to dealing with natural language can be likened to a person who is hard of hearing. Even though such a person does not always "catch" all the words someone speaks, he can usually guess the meaning of a sentence on the basis of his knowledge of the general topic of conversation and the few words he did hear clearly (including the ordering and context of those words).

The compuational model used in the translation process is that of a nondeterministic finite state automaton (Figure 2). Based on the current state of the automaton and the input class assigned when a term is encountered, the appropriate entry in the state table determines any addition to be made to the intermediate representation of the request, and also establishes the next state to be entered by the automaton. Each state of the automaton can be associated with a partial understanding of the request. This understanding is based on that portion of the request which has been analyzed up to that point. If the translator reaches a dead end in its search for a meaningful interpretation of the request, it backs up to the previous term and looks in the state table for an alternative interpretation. This process is continued until all choices are exhausted or a consistent interpretation has been found. Using this approach eliminates the need for storing a grammar of English (saving a considerable amount of memory) and allows the translator to handle ungrammatical or partially understood inputs.

The intermediate representation is a nesting of function calls to routines in the request processor. The possible functions are shown in Figure 3. The basic idea is that most requests have a simple syntax; one section indicating the type of information desired, and the other a series of specifications limiting the domain of interest. The functions can then be divided into two groups: those specifying a particular lesson

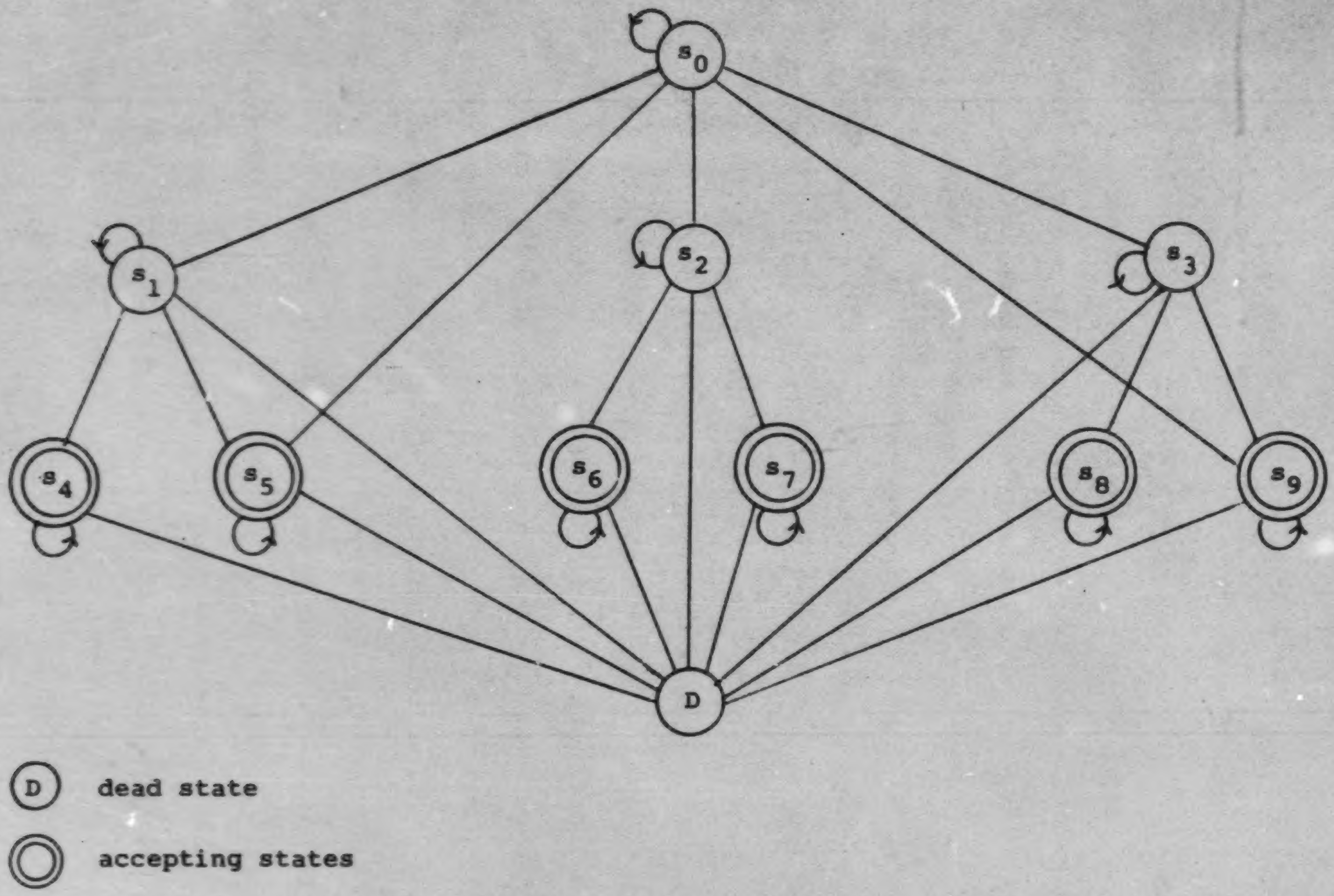


Figure 2: The State Diagram of the Non-deterministic Automaton

or set of lessons; and those for particular types of information, having as arguments a specification function, or a nesting of type and specification functions. This is discussed more fully in a Ph.D. thesis by Pradels [2].

4.5. Paraphraser

The paraphraser produces a paraphrase of the original request based on the intermediate representation produced by the English translator, allowing the student to confirm whether his request has been properly understood by the system. If so, he can proceed to the response. If not, he can immediately rephrase his request. (Also, in many cases, he can deduce what caused the system to misunderstand his request.)

4.6. Request processor

The request processor accomplishes the task of transforming the intermediate representation of a request into a specification of the type of response to be generated. By analogy with the output of the English translator, this specification has been called the "intermediate form of the response".

Several simple heuristics are used in the request processor, based on the principle of determining as quickly as possible which area of the database contains the answer to the original request, and what possible response of the system will display that area of the database. In some cases, the request processor simply indicates to the response generator the area of the database to be displayed (for example, the term number of a course record). In other cases, the request processor assembles some data from the database and passes that information to the response generator (for example, a list of term numbers or lesson records which match a given specification).

Specification functions:

These functions return a set of lessons depending on the value of their variables.

LN : has two arguments, a course name and a lesson name. It returns the lesson defined by these.

LS : returns the set of lessons which are defined by characteristics other than their names. These characteristics might be a Boolean list of keywords, a type (lesson, exam), a course to which they belong, an author name, a level of difficulty, a sequence specification, a time period, or other specifications, such as whether the lessons have already been taken or not. Any of these characteristics can be specified or negated.

Information type functions:

TF : tests if the set is empty or not

AB : returns abstracts of its elements

AN : returns names of author of its elements

NB : returns the size of the set

GR : returns the grade required by the instructor

GO : returns the grade obtained by the student

TS : returns the schedule to achieve relative to the elements of the set

TT : returns the schedule achieved by the student

PQ : returns the set of lessons which are prerequisite to the argument set

SQ : returns the set of lessons which are sequels to the argument set

SI : returns the set of lessons which are similar to the argument set

BE : has two arguments, a lesson and a set. Tests if the lesson belongs to the set.

Figure 3. Request processing functions

4.7. Response generator

The response generator accomplishes the task of transforming the intermediate form of the response into a display which can be presented to the user. There are four basic types of response which will be discussed below: list of lessons, record display, graphics display, and feedback message.

4.7.1. List of lessons

The response to a large number of requests presented to the GUIDE is a list of lessons matching a given specification. For this response, the response generator simply produces a listing of the name and abstract of the lessons which have been retrieved.

4.7.2. Record display

Record displays are generated for lesson, course, and student records. In the original design of the response generator, it was intended that most requests would receive a prose response. As an intermediate step in the development toward that end, it was decided to display the entire record which contained the piece of information which had been requested. This approach to the response proved to be so successful that implementation of the prose approach was abandoned.

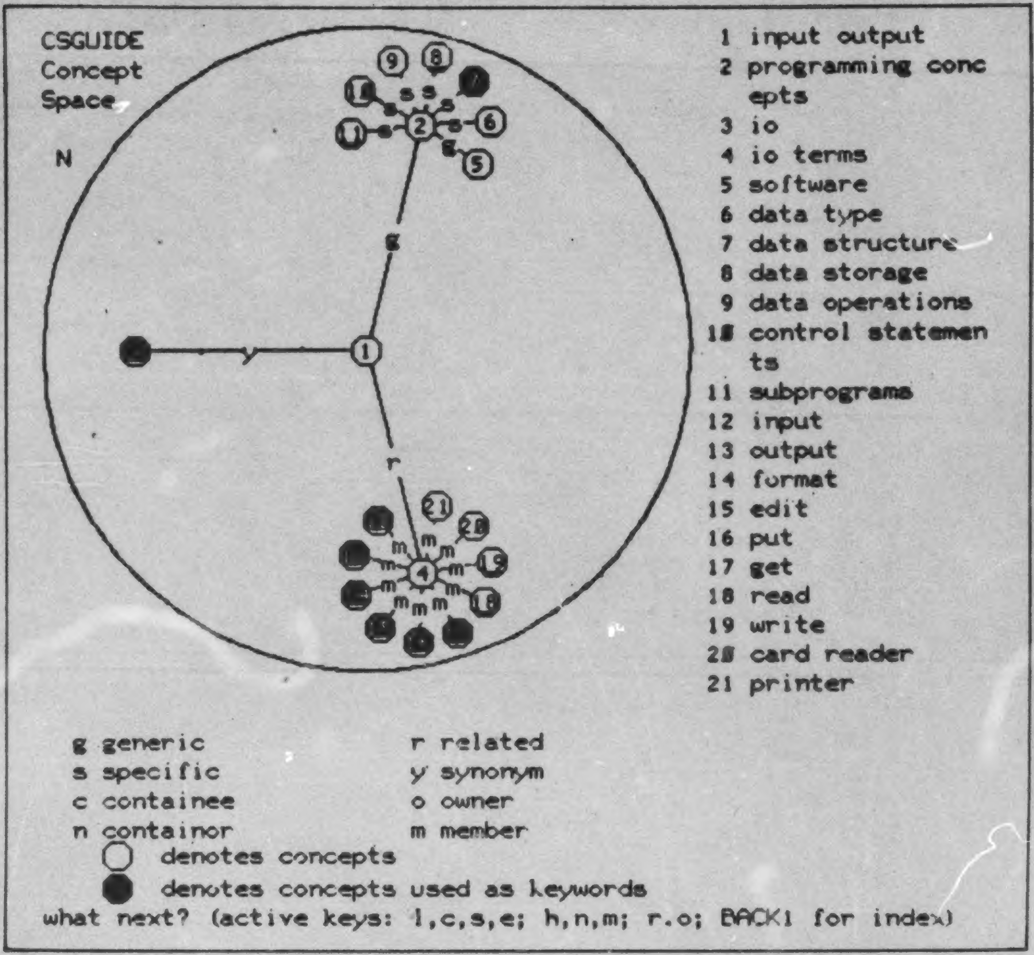
With this approach, we anticipate in advance a whole sequence of potential questions (hence reducing total CPU usage) and are able to properly answer a large class of poorly-phrased questions. This allows the student to type shorter requests and still obtain the desired information.

4.7.3. Graphic displays

One of the challenging research tasks in implementing the GUIDE was the development of an effective means of communicating the structure and content of the concept space. This network possesses a very rich structure of interrelationships which is difficult to describe. Fortunately, the PLATO terminal provides a graphics capability which helped solve that problem. The GUIDE utilizes three different types of graphical displays to help present different points of view of the concept space: the neighborhood, hierarchical, and mixed mode displays.

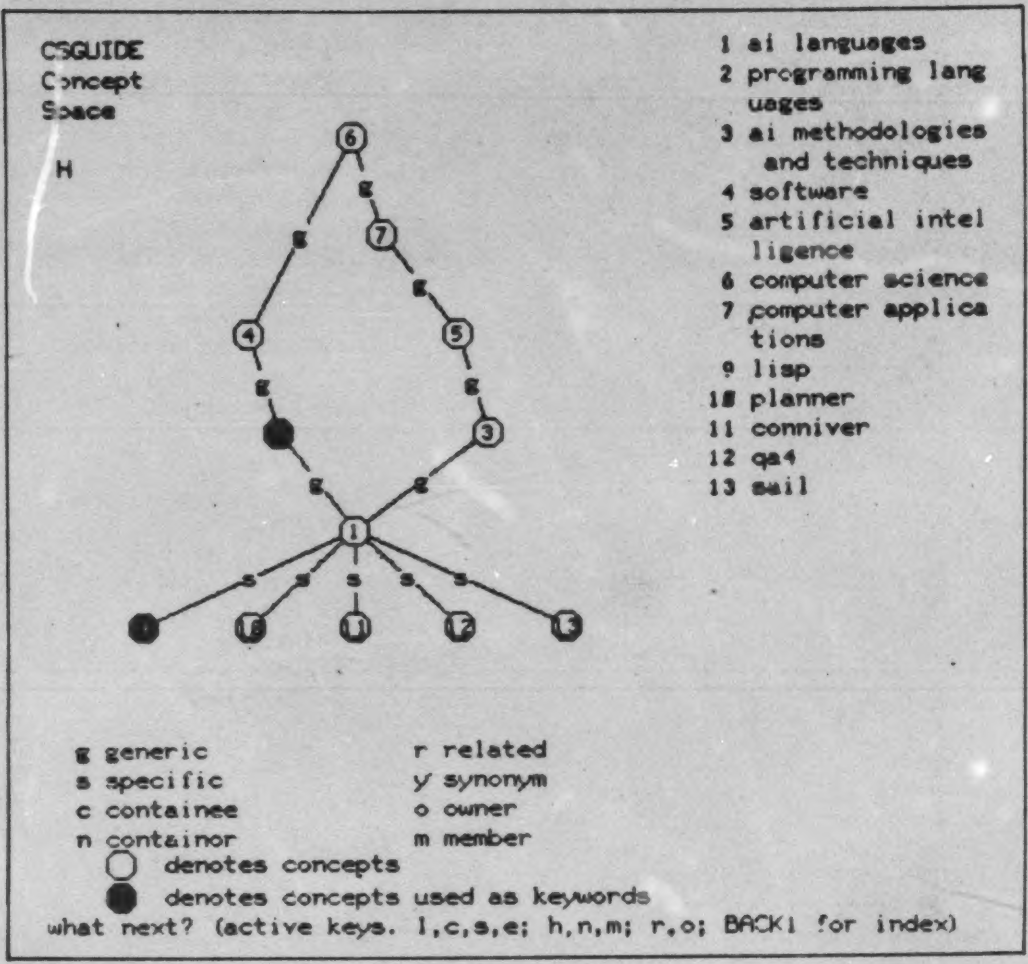
The neighborhood display shows the concepts which lie in the immediate neighborhood of a given concept. Figure 4 shows a sample neighborhood display. The first circle of nodes shows the "first generation" of concepts--those that are directly related to the central concept of the display. The secondary circles of nodes show the "second generation" of concepts--those that are directly related to the first generation concepts (and hence are two generations away from the central concept). Exploration of the concept space can be accomplished by requesting successive displays where the central node in each new display has been selected from the first or second generation of the previous display.

Whereas the neighborhood display gives a sense of "distance" in the concept space, the hierarchical display imparts a sense of "direction". Figure 5 shows a sample hierarchical display. Basically, the hierarchical mode enables one to traverse the classification tree, pursuing topics by narrowing or expanding one's scope of interest. The sense of direction imparted by the hierarchical mode is how "high" or "low" a given concept



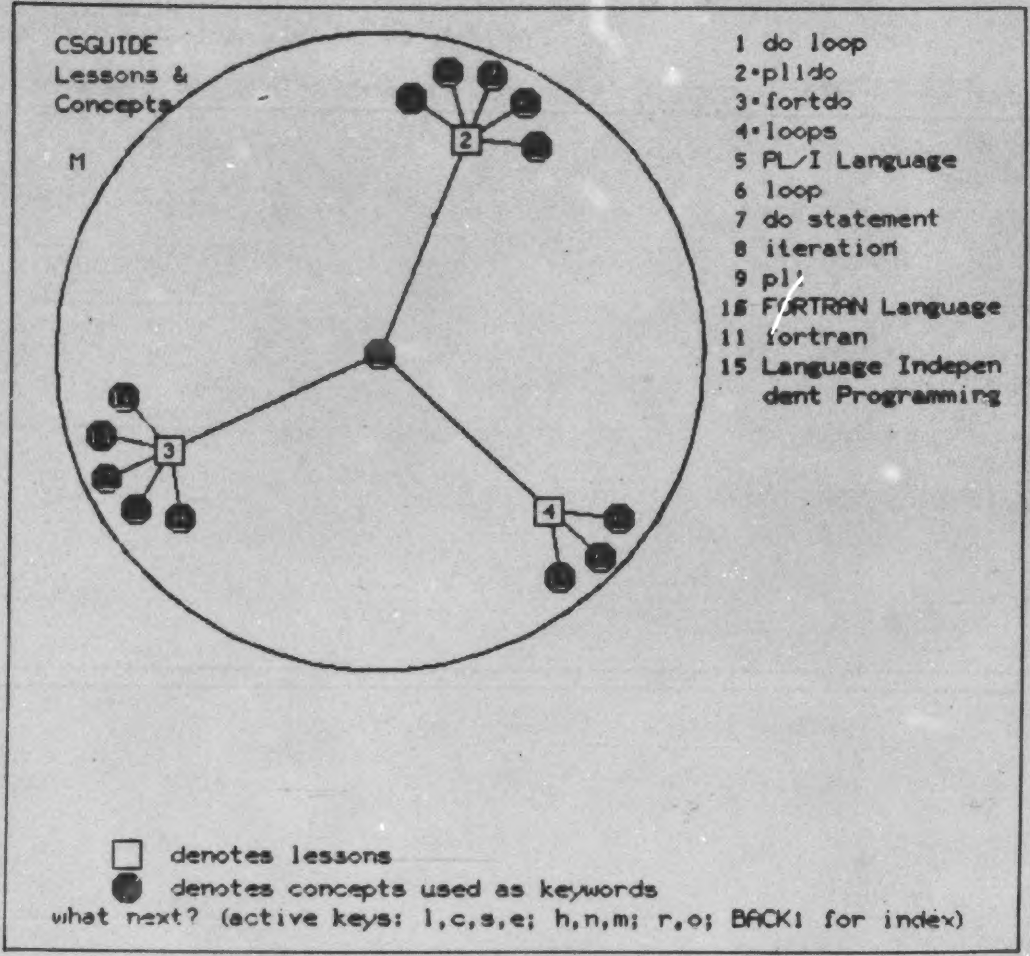
```
1 input output
2 programming conc
  epts
3 io
4 io terms
5 software
6 data type
7 data structure
8 data storage
9 data operations
10 control statemen
  ts
11 subprograms
12 input
13 output
14 format
15 edit
16 put
17 get
18 read
19 write
20 card reader
21 printer

g generic              r related
s specific             y synonym
c containee            o owner
n containor            m member
  ○  denotes concepts
  ●  denotes concepts used as keywords
what next? (active keys: l,c,s,e; h,n,m; r.o; BACK1 for index)
```

Figure 4: Sample Concept Space Neighborhood Display



```
CSGUIDE                                   1 ai languages
Concept                                   2 programming lang
Space                                       uages
                                          3 ai methodologies
 H                                           and techniques
                                          4 software
                                          5 artificial intel
                                            ligence
                                          6 computer science
                                          7 computer applica
                                            tions
                                          9 lisp
                                         10 planner
                                         11 conniver
                                         12 qa4
                                         13 sail

   g generic              r related
   s specific             y synonym
   c containee            o owner
   n containor            m member
     ○   denotes concepts
     ●   denotes concepts used as keywords
what next? (active keys. l,c,s,e; h,n,m; r,o; BACK1 for index)
```

Figure 5: Sample Concept Space Hierarchical Display


CSGUIDE
Lessons &
Concepts

M

1 do loop
2•pl1do
3•fortdo
4•loops
5 PL/I Language
6 loop
7 do statement
8 iteration
9 pl1
10 FORTRAN Language
11 fortran
15 Language Independent Programming

☐ denotes lessons
● denotes concepts used as keywords
what next? (active keys: l,c,s,e; h,n,m; r,o; BACK1 for index)

Figure 6: Sample Concept Space Mixed Mode Display

is in the classification tree, and where it is relative to the "root" of the tree--the concept "computer science."

The mixed mode display was introduced to facilitate the graphical presentation of the set of lessons which are attached to a given concept. It is called "mixed" since both lessons and concepts appear in the display. An example of such a display is shown in Figure 6. An interesting interpretation of this display is that in a very real sense, a lesson can be viewed as a relation in the concept space, providing a link between various concepts. For example, in Figure 6, the concept "do loop" has the "fortdo" relation with the concept "fortran."

4.7.4. Feedback response

A feedback response is presented to the user asking for clarification or further information in three situations: when an ambiguous term is used, when the translator can't find a valid interpretation of the request (often caused by a significant word of the request not being in the database), or when the request processor does not have sufficient information to process the request (e.g., no student record available).

References

[1] Eland, D. R. An information and advising system for an introductory computer science course. Report UIUCDCS-R-75-738 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, June 1975.

[2] Pradels, J. L. The Guide, an information system. Report UIUCDCS-R-74-626 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, March 1974.

5. Interactive test construction and administration in the generative exam system (L. R. Whitlock, R. I. Anderson)

5.1. Introduction

The Generative Exam System is a completely interactive system for the construction and administration of examinations. Since all tasks associated with examinations (from exam writing through analyses of exam results) are handled interactively in the system, the Generative Exam System offers many advantages over written exams. These advantages include a considerable savings in time and expense in writing, duplicating, and grading exams; exam security, provided by the fact that each student receives slightly different questions; consistent and accurate exam grading; the capability of allowing each student to review the scores and correct answers on his exam immediately after he finishes it; and the immediate availability of a complete analysis of exam results after a class finishes an exam.

The operation of the Generative Exam System is described in detail in one document (1), and the development and evaluations of the system are described in detail in another (2). Some of the major aspects of this project are outlined below.

5.2 System organization

The exam system differentiates between two kinds of users - student and instructor. An instructor has access to both student and instructor options, while all other users have access to the student options only.

Figure 1 is a block diagram of the major components of the Generative Exam System. All users enter the system through the Monitor,

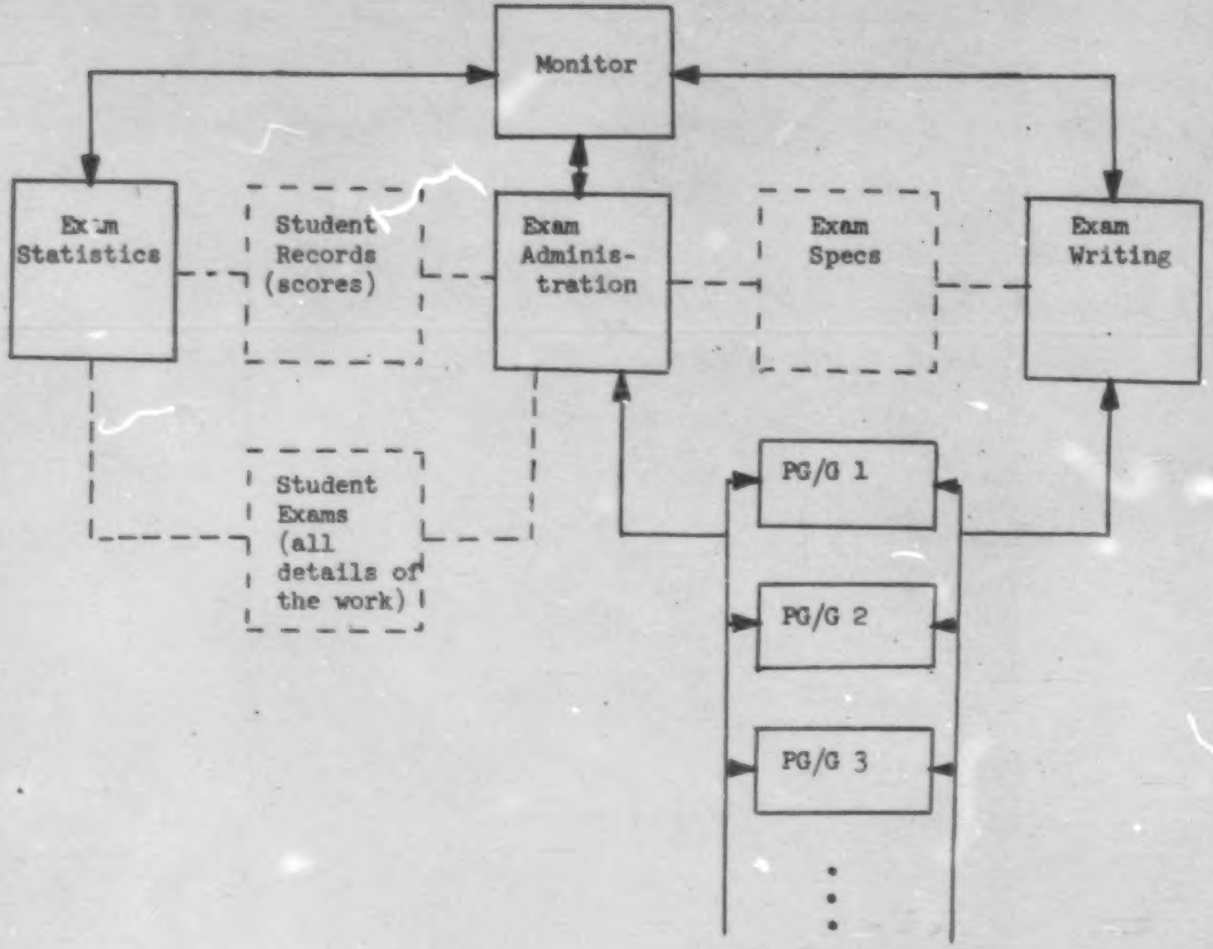


Figure 1: Block Diagram of the Major Components of the Generative Exam System

and on initial entry are allocated a record in the Student Records data base and a permanent storage area for their work in the Student Exams data area. Instructors write exams in the Exam Writing section by writing problem specifications for each desired problem generator/grader (pg/g). This set of problem specifications is assembled into an exam specification and stored in the Exam Specs data area. When a student takes an exam (in the Exam Administration section), the appropriate exam specification is transferred from the Exam Specs data base to the student's permanent storage area in the Student Exams data base. The same area is used to record his work as he changes from problem to problem. Instructors may review exam results in the Exam Statistics section.

The heart of the system is the set of pg/g modules which produce examination problems. Each pg/g is an independent module which handles all aspects of one problem except data storage. These functions of a pg/g include guiding an instructor through the process of writing problem specifications, generating problems (under the constraints of the problem specifications), administering problems to students, and reviewing problems with students after their exam.

Since each student's problems are generated as he takes his exam, there is no pre-test security problem. The generation schemes used by the problem generator/graders are designed to operate under time and storage space constraints so that delays and distractions to the student are avoided. The generation schemes produce a large number of similar problems by randomly generating numbers and character strings and assembling problem pieces into complete problem structures. Some pg/g's have the capability of generating problems at different specified levels of difficulty in their subject areas.

The problem generator/graders employ grading schemes which award credit for partially correct responses by checking responses for variants of the correct answer or by grading the correctness of one response on the assumption that the previous response in that problem is correct. An example of the former grading scheme is found in the FORTRAN expressions pg/g. If the correct answer to an expression were "-45.0", the pg/g would award partial credit for the responses "45.0", "45", or "-45". The DO-loop pg/g uses the second grading scheme mentioned above. The response given by a student for each iteration of the DO-loop is compared to the correct answer for that iteration and to the answer calculated from the student's previous response. Full credit is awarded if the response agrees with either answer.

5.3. Experimental results

Two experiments were conducted to evaluate the Generative Exam System. In each experiment, subjects were administered an exam on PLATO and a written exam. The coefficients for the PLATO exam scores correlated with the written exam scores averaged .64 in one experiment and .60 in the other. Assuming that the written exams gave valid measurements of each student's knowledge, these results suggest that exams in the Generative Exam System are as effective at evaluating students as written exams.

The experiments also studied the tailored style examination. In a tailored exam, the difficulty levels of the problems are altered as the student works through the exam in an attempt to match the problem difficulty level to the student's level of knowledge. This approach should more accurately measure the extent of a student's knowledge and make this measurement in less time and with less frustration to the student than the tradition style examination. A tailored exam would be useful in criterion-referenced grading situations such as self-paced courses.

In the experiments conducted to evaluate the Generative Exam System, some subjects took tailored PLATO exams and other subjects took regular PLATO exams. (Regular PLATO exams are very similar to written exams.) The coefficients for the PLATO exam scores correlated with the written exam scores were higher for the group of subjects who took tailored exams than any other PLATO exam group (.83 for the tailored subjects versus an average of .59 for the subjects who took regular exams in one experiment, and .68 for the tailored subjects versus an average of .53 for the subjects who took regular exams in the other experiment). These results indicate that the tailored exam idea is at least as effective in evaluating students as regular style exams. However the implementation of the tailored exam in the Generative Exam System was inefficient in terms of time (tailored subjects spent an average of 40.32 minutes on their exam as opposed to an average of 31.78 minutes for the other subjects) and was unpopular (as indicated by questionnaire results). Improvements to the Generative Exam System which could make tailoring more efficient and less unpopular have been planned.

The studies conducted with the Generative Exam System suggest that interactive exams are useful and effective in evaluating students and merit continued research, especially in the areas of problem generation and grading and tailored exams.

5.4. The quiz system

In an effort related to the exam system, a special quiz system has been developed which enables presentation of a criterion-referenced quiz following a PLATO computer science lesson. Designed and implemented by R. I. Anderson from a concept proposed by R. G. Montanelli, the system is intended 1) to provide a student taking a PLATO computer science lesson with both a means to assess how well he or she learned the material that the lesson is intended to cover and a tool to aid in learning the topic at hand, and 2) to provide members of the ACSES staff with a means to assess how effective and thorough a PLATO computer science lesson is at teaching its topic.

5.4.1. Quiz system operation

The system consists of a quiz system monitor and a pool of PLATO quizzes available for administration to students at the conclusions of individual computer science lessons. Each quiz has been designed to pertain to some well-defined topic within the computer science field, and each quiz question has been selected by the quiz author to test pertinent details of the topic's content. Incorporated into each quiz is a data collection facility to record students' question responses.

Access to the pool of quizzes is provided, via the quiz system monitor, to instructors who wish to use the quiz system. The system monitor allows such instructors to select a quiz of the desired content area, interactively design the quiz to best suit the particular lesson's needs, view the quiz exactly as a student will, and finally "attach" the quiz to the chosen instructional lesson. This latter task requires minor changes be made to the code of the instructional lesson to enable an interface with the quiz. These changes are clearly outlined to the user when quiz attachment is arranged.

Figure 2 illustrates how the instructional lesson/quiz interface operates. At the time a quiz is to be administered to a student, control is transferred from the instructional lesson to a quiz system program that functions as a link to the quiz (arrow A). This program determines which quiz of those available in the system is to be presented for this particular instructional lesson, and control is transferred to it (arrow B).

Interaction between the system program and the lesson that produces the quiz occurs at various points during quiz administration (arrow C). Since all quizzes are designed with a similar structure, most aspects of student-quiz interaction are uniform across quizzes. The basic sequence is as follows:

1) Once the quiz system program verifies the existence of a quiz for a particular instructional lesson, a page detailing the quiz's purpose is presented to the student.
2) Quiz questions are then administered; questions may be skipped if the student so desires, and all questions may be reanswered in case an error was made.
3) When the student decides that his or her attempt at the quiz is complete, he or she may advance to the presentation of the corrected quiz, which is accompanied by clarifying explanations.
4) Lastly, the student is informed of his or her final quiz scores as well as the average score received by others in his or her course.

Following a review of the corrected quiz, a student is returned to the system program a final time (arrow D) where return to the appropriate instructional lesson is provided (arrow E).

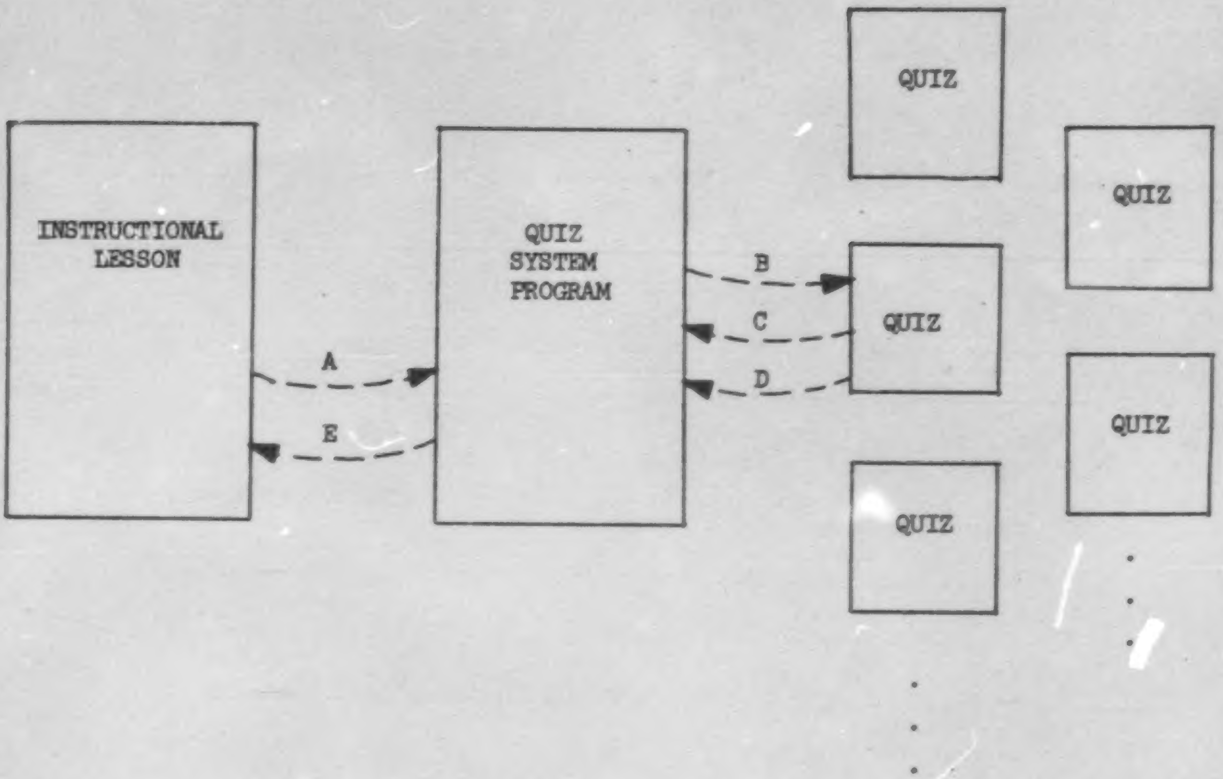


Figure 2: Simplified View of Interactions
During Quiz Administration

Once a quiz has been attached to an instructional lesson, each student taking that lesson will also take the quiz, and be informed of the average score obtained on the quiz by other members of his or her course. Each instructor will have access, via the quiz system monitor, to data that is accumulated for each quiz question and will be informed of both the average quiz scores and the average amount of time needed for taking the quiz for all courses that have used the associated lesson. Each authorized ACSES staff member will also have access to the data accumulated for each quiz question, to facilitate analysis of the effectiveness and completeness of both the instructional lesson and the quiz.

When course instruction is completed, instructors detach the quizzes from the associated lessons. This procedure, which can be accomplished through the quiz system monitor, clears the accumulated course data. During the time that a quiz is detached from an instructional lesson, no reversal of the quiz-accomodating lesson alterations made earlier are necessary. Transfer of control to the quiz system program to administer the quiz (arrow A of Figure 2) will simply result in the display of the message: "No quiz currently exists for this lesson". Control is then immediately returned to the instructional lesson (arrow E).

5.4.2. Past experience and current status

Currently, only quizzes pertaining to selected topics of the FORTRAN programming language have been implemented. As the first quizzes of the system, these were all designed and developed from objectives used to develop the existing FORTRAN instructional lessons; thus instructors had very little opportunity to manipulate the quiz design to satisfy other lessons' needs. More quizzes are being developed, however, and existing quizzes are continually being improved. The availability of a quiz that suits a user's lesson's needs may be investigated via entrance into the quiz system monitor.

The initial trial of the quiz system occurred during the fall semester of 1975, when a FORTRAN character manipulation quiz was presented following an instructional lesson on the same topic. Quiz question responses accumulated from students in an introductory computer science course clearly indicated various deficiencies within the instructional lesson and even revealed an instructional error. The lesson was thus restructured, the discovered error was corrected, and the identified deficiencies were eliminated.

Subsequent use of the quiz system occurred during both the spring semester and the summer session of 1976. Four quizzes were presented following appropriate instructional lessons to students in four different introductory computer science courses. Preliminary analysis of data accumulated by these administrations indicated shortcomings both in instructional lessons and in quizzes. Corrective action is currently underway.

References

[1] Whitlock, Lawrence R. Documentation on the generative exam system. Unpublished memo, Department of Computer Science, University of Illinois at Urbana-Champaign, June 22, 1976.

[2] Whitlock, Lawrence R. Interactive test construction and administration in the generative exam system. Report UIUCDCS-R-76-821, Department of Computer Science, University of Illinois at Urbana-Champaign, September 1976.

[3] Anderson, Richard I. User's manual and guide to the ACSES quiz system. Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, September 1976.

6. Automatic judging of student programs (R. L. Danielson, P. Mateti, W. D. Gillett)

An automated system for instruction should be capable of making judgements and providing comments on student programs, analogous to the role played by teaching assistants and graders in the more traditional means of instruction. Our efforts to provide this capability have resulted in two lessons which ask the student to write fairly sophisticated programs and attempt to judge these programs interactively with respect to both correctness and good design, and a categorization of anomalies in beginning students' programs which are detectable by automatic analysis routines.

A program by R. Danielson exposes students to a dynamic example of the top-down programming process by monitoring their attempts to write a PL/I program for symbolic differentiation of a polynomial. PATTIE (Programmed Aid for Teaching Top-down programming by Interactive Example), mimics the action of a human tutor, in that she engages the student in an interactive dialog, judging the correctness of student-suggested refinements and providing hints and comments where necessary. The tutor uses an AND-OR graph as a model of the stepwise refinement process, which student and tutor traverse together in the course of program development. Danielson (1975a, b) discuss this tutor in detail.

The other lesson is a sorting laboratory and program verification system developed by P. Mateti. This system allows the student to write an arbitrary in-place sorting program in a programming language with specially designed sorting primitives. A special interpreter then provides a dynamic display of the status of the array and indices during execution. In addition, the student may provide assertions about the state of the keys in the array, and the truth or falsity of these assertions is indicated during execution. The student may submit completed programs to the program verification routines, which use the inductive assertion method to prove the program correct, or prove it incorrect and provide a counterexample. A special theorem prover, which is highly efficient in this restricted domain, is the heart of the system. A full description is in Mateti (1976).

Finally, a study is being conducted by W. Gillett aimed at determining and categorizing various legal programming constructs whose presence in a program probably indicates a lack of understanding by a student (e.g., B**1/2, which is equivalent to B/2). The idea is to determine techniques by which such errors may be detected; the general approach is to use iterative analysis methods on a flow graph equivalent to the student's Fortran source program. An automatic program containing such techniques, while being unable to direct the student toward correctly developing a program because it isn't aware of the algorithm being implemented, would still be able to provide incisive comments on improving program efficiency, correctness, and understandability.

The following subsections provide a more detailed discussion of these three efforts.

6.1. PATTIE

Top-down programming provides a means for the programmer to restrict the scope of the problem he must solve to a manageable level. The principal aid in this restriction of scope is the use of levels of abstraction. Successive refinement begins with an abstract description of the task to be accomplished. This task is then refined, that is, described as a series of slightly more specific tasks which, when combined, solve the problem. Each of these tasks at this second level is refined in turn, producing a third level of task descriptions, and the process continues until tasks have been described in sufficient detail to be easily translated into programming language statements. Task descriptions commonly employ a mixture of natural language and programming language statements, which allows much of the complexity of the programming language to be ignored until needed. The successive levels of task descriptions allow the programmer to concentrate most of his attention on the task he is currently refining, and yet be sure of the proper integration of that task with the whole solution.

There must be three separate aspects of a system designed to tutor a student about top-down programming. First, because the tutor must monitor the process of developing a program, it is necessary to provide a representation for acceptable methods of solving a problem, as well as acceptable completed solution programs. Second, because we want the student to learn something about the technique of the top-down programming, the tutor must have some instructional strategy to aid this learning, and use this strategy in interacting with the student. Finally, the importance of natural language to the successive refinement process requires the tutor possess some natural language capability sufficient to understand suggested refinements and allow them to be related to the knowledge of acceptable solutions.

Let's look at each aspect in further detail.

6.1.1. Representation of knowledge

Any sort of problem solving activity (such as programming) involves reducing the original problem to one which is understood and can be solved, using some rules or problem reduction operators. Problem solving tutors for other subject areas (simple integration, logic theorems) give the student a wide range of experience, and are capable of handling a correspondingly wide range of both prestored and student-suggested problems. To accomplish this, heuristic problem solving routines, with capabilities similar to those of the students being tutored, are integral parts of the system. Such an approach is possible due to the quantitative nature of the subject areas. Solving problems in integration or proving simple logic theorems requires using only a small number of rules applicable to many problems.

Unfortunately, in top-down programming there is no small set of general rules which can be applied in many situations. There is, instead, a very large number of distinct refinements which are applicable in only a small number of instances. This, coupled with the difficulty of clearly establishing a new problem state following a reduction expressed in natural language, led us to explicitly store knowledge about the exact solutions to a particular problem, and change this knowledge to allow the tutor to accommodate other problems. This leads to a need to represent a top-down solution.

The traditional representation for the stepwise refinement process is a tree. The root represents the initial problem to be solved, leaves represent statments in the target programming language, and each intermediate node represents a subtask on one of the levels of abstraction. Such a tree, however, represents only one solution; there are likely to be many correct solutions to any particular problem. Hence we decided to represent the solution knowledge as an AND-OR grah.

The basic idea behind an AND-OR graph is reducing a problem to a series of subproblems, just as in stepwise refinement. In such a graph, each node represents a problem. Solving the problem represented by an AND node can be accomplished by solving all the subproblems represented by the successor nodes. Solving the problem represented by an OR node can be accomplished by solving any one of the subproblems represented by the successor nodes. The solution to the initially given problem (represented by the root of the graph) is successively reduced to the solution of sets of subproblems, some of which might be immediately recognized as being solved (LEAF nodes), others of which might need further reduction. Intuitively, an OR node corresponds to a point in the development of a solution where a choice must be made between several (equally correct) approaches. An AND node represents a point at which refinement involves several tasks which must all be done to solve the problem. Figure 1 is a small portion of the AND-OR solution graph for the problem the tutor is currently using: developing a program for the symbolic differentiation of a polynomial.

In order to use an AND-OR graph as the basis for a tutor of successive refinement, several features were added. As Figure 1 shows, branches between nodes are tagged with English phrases ("transition phrases") which are usually descriptions of the tasks represented by the node each branch leads to. Thus, nodes represent subproblems to be solved, and brances are tagged with English descriptions of the subproblems they lead to. PATTIE uses these transition phrases to determine the path the student is taking through the graph. Other branches (leading to LEAF nodes) are tagged with PL/1 statements and represent the final

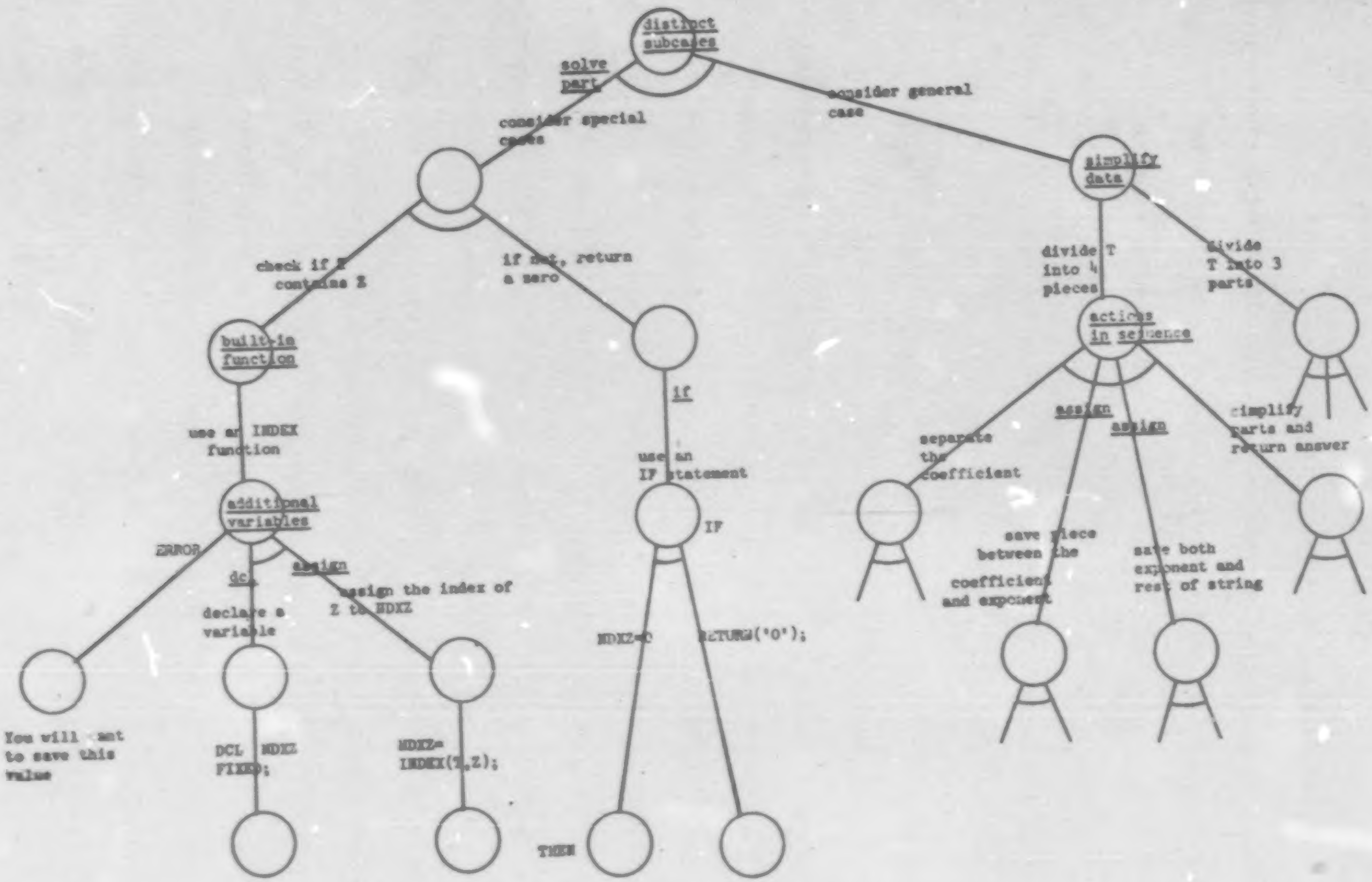


Figure 1: Partial solution graph

step in refinement of a particular task, namely translation into the target programming language.

The remaining features added to the basic AND-OR graph formalism are special branch or node types. Special ERROR branches allow the problem expert who develops the graph to provide specific hints to be given to the student only in certain contexts. These ERROR branches may run from either AND or OR nodes, and lead to LEAF nodes which have error messages attached. ERROR branches tagged with transition phrases ("expected" ERROR branches) correspond to bad approaches the problem expert felt students were likely to attempt at that point, and the error message can explain why that approach is not good. Untagged ("universal") ERROR branches lead to error messages which simply suggest explicit actions which are probably needed at that point.

A second special branch type was needed to handle the common practice in top-down program development of intermixing partial programming language statements with English descriptions of refinements. This branch type may be tagged with PL/1 statements and marked so as to be displayed as soon as the node is encountered, but not traversed.

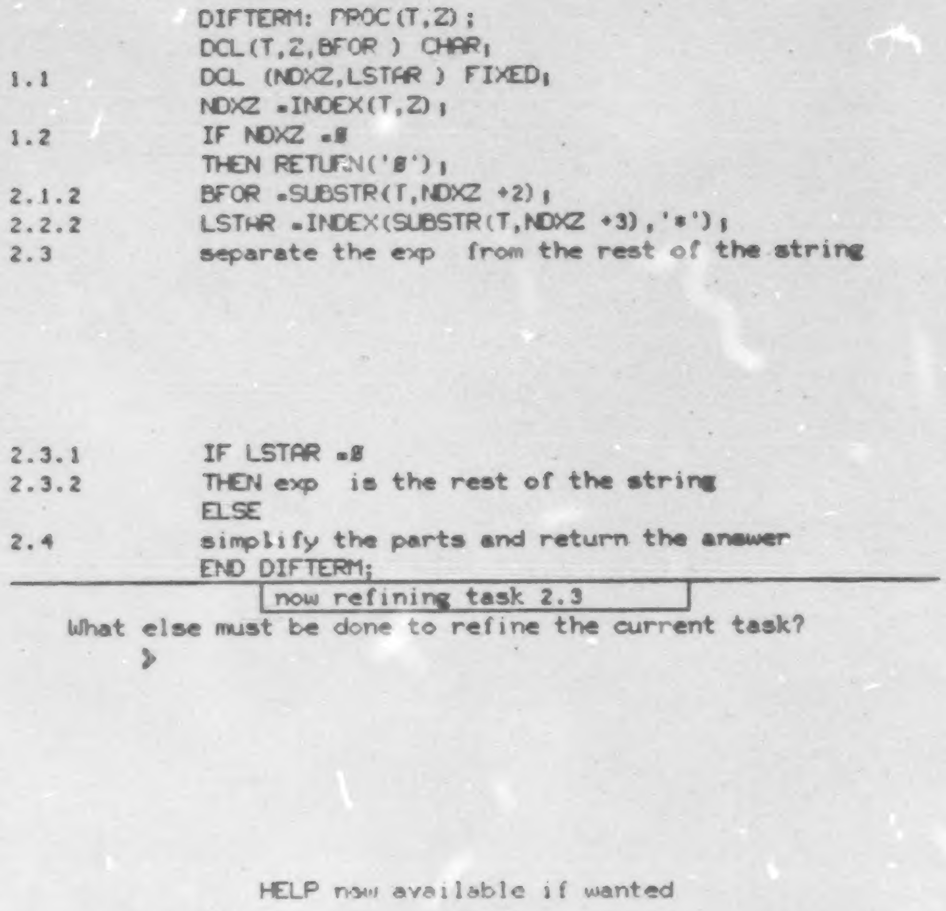
The final special feature is the PROC node, which allows invocation of a subroutine before it is programmed in detail. When encountered in the refinement of one procedure, the PROC node acts like a LEAF, but also causes the interaction control program to stack the PROC as the root of the new procedure's subgraph, for later detailed development.

6.1.2. Student-tutor interaction

In relation to the AND-OR graph, the successive refinement process corresponds to tracing a path through the graph from a root to some subset of the LEAF nodes which represent a solution program. The exact path taken is determined by suggested refinements input by the student. At any given time, the student is actively refining only one task, the "current task." This current task is represented in the refinement graph by a single node, which PATTIE determines the correctness of student-suggested refinements by matching them against branches in the graph at and below the current node. Once the student has described all the actions needed to refine the current task, a new current task is selected by simply traversing one of the described branches from the current node to a new current node. The order in which these branches are traversed is determined by the control program using a depth-first traversal algorithm.

The means by which PATTIE may interact with the student is the display screen of the PLATO IV terminal. Figure 2 is a copy of the screen as the student sees it. The upper 20 lines are the "program area" and contain the developing solution program. The lower part of the screen is the "scratchpad," the area where the dialog is conducted.

On the left-hand side of the program area are a series of "task names" indicating the relationship of each task to others in the solution, exactly as the relationships between sections of this thesis are described by the section numbers. Task names are assigned when the refinement task is initially described, based on the task name of the current node and the current node type. Each refinement at an AND node receives a task name composed of the task name of the AND and a suffix



```
                DIFTERM: PROC(T,Z);
                DCL(T,Z,BFOR ) CHAR;
1.1             DCL (NDXZ,LSTAR ) FIXED;
                NDXZ =INDEX(T,Z);
1.2             IF NDXZ =0
                THEN RETURN('0');
2.1.2           BFOR =SUBSTR(T,NDXZ +2);
2.2.2           LSTHR =INDEX(SUBSTR(T,NDXZ +3),'*');
2.3             separate the exp  from the rest of the string




2.3.1           IF LSTAR =0
2.3.2           THEN exp  is the rest of the string
                ELSE
2.4             simplify the parts and return the answer
                END DIFTERM;
------------------------------------------------------------
                   | now refining task 2.3      |
   What else must be done to refine the current task?
       >



                   HELP now available if wanted
```

Figure 2: The student's screen display

indicating the number of the branch matched by the refinement. At OR nodes, on the other hand, the task name of the refinement is simply that of the OR itself, since only one branch leaving the node is ever traversed. These task names indicate the relationship between tasks described in the successive refinement process.

Within the program area there are three distinct subareas. At the very top are programming language statements, corresponding to refinements which had been described well enough to be translated and displayed as code by PATTIE. Immediately below these is the natural language description of the current task. Finally, at the very bottom of the program area are other tasks awaiting further refinement. This is essentially a stack of refinements described at AND nodes during solution development, but whose corresponding branches have not yet been traversed by the control program. These refinements provide a context in which the student can devise his refinements for the current task, but they needn't yet be considered in detail.

The scratchpad area is where PATTIE accepts student inputs, displays hints, or reveals the anticipated refinements once available hints have been exhausted. This interaction goes on (solely in the scratchpad area) until a correct refinement for the current node is input by the student. At this point, the refinement task description is moved to the program area, the exact location depending on the current node type.

If that current node is an OR, the student only needs to input a single refinement which matches one of the branches leaving the node, and the control program's actions are correspondingly simple. It simply determines which branch leaving the node is matched by the student input, replaces the current task description on the screen with the suggested refinement, and traverses the matched branch to a new current node.

AND nodes, on the other hand, correspond to points in the solution process where refining a task requires describing several separate subtasks. As each refinement is accepted, its description is moved to the program area stack. When all needed refinements have been described, the current node is pushed on a stack of active AND nodes (i.e., nodes with all necessary refinements described, but at least one branch leaving the node untraversed), the top refinement description on the program area stack is moved up to become the new current task, and the leftmost branch from the node is traversed. This corresponds to a depth-first traversed of a particular path through the solution graph.

Of course, an essential part of the tutorial process is providing hints to the student when he makes a mistake. There are several levels of prompts coded into the dialog routines which provide slight hints to the student. These are dependent on the current node type and may be superseded by more explicit hints contained in the solution graph, if such are available. Such specific hints may be provided by the problem expert who develops the graph by means of ERROR branches. If an expected ERROR branch leaves a given node, the error message the branch leads to will be displayed only if a student input matches the attached transition phrase when that node is the current node. ERROR branches are not examined during the lookahead matching process, to avoid potentially misleading hints. If the branch is a universal ERROR branch, the error message is displayed in response to the first wrong input received when that node is the current node, and then the prompt sequence described above takes over.

Finally, the tutor contains a student model based on a list of semantic concepts relative to problem solving and the subject area of the particular problem. Each node and branch in the solution graph may be tagged with one of these concepts, and for each concept the model keeps track of the probability that the student will suggest a correct refinement at a point in the graph tagged with that concept. This information can be used to provide additional hints to the student, or to modify the standard procedure of asking for suggestions and immediately display one or more of the desired refinements.

6.1.3. Natural language capability

An analysis of protocols between a human tutor and a student over the same programming problem the tutor is concerned with indicated two things:

(1) student utterances are short, ungrammatical, and relatively isolated from each other;

(2) students use only a small number of patterns in their utterances (both typed and verbal)

Item (1) ruled out the use of a linguistic-based understanding system, and item (2) provided hope that the tutor could make do with the simple dialog understanding system provided by the PLATO IV author language, TUTOR.

Essentially, this facility is a keyword recognition, pattern matching scheme. An author specifies a vocabulary, consisting of a number of disjoint classes of synonymous words (groups of "content" words) and a list of words which are allowed in a student's inputs but which carry no meaning ("ignorable" words). Elements of a synonym class may be single words or "phrases," which are a series of two or more words which must appear contiguously. Phrases provide a simple means of handling common idioms, and may consist of ignorable words, content words appearing elsewhere in the vocabulary, or completely new words.

TUTOR's facility attempts to assign a meaning to typed inputs by matching the input against a series of stored patterns. Each pattern consists of representatives from one or more of the classes of synonymous words in the vocabulary. Since there are usually many ways of expressing an idea in natural language, it is frequently necessary to attach more than one keyword pattern to a single "meaning list." For example, since keyword order and number of keywords are important in a pattern match, if it was desired to assign the same meaning to the inputs "a brown cat," "a cat that is brown," and "a cat" (assuming "cat" and "brown" are content words and other words are ignorable), the meaning list must include the patterns "brown cat," "cat brown," and "cat."

One of the biggest drawbacks of a synonym-class approach such as this is that a word can have several different meanings in different contexts. Since no word can be in two classes (except as part of a phrase), classes which contain the same words must be coalesced. This introduces a certain amount of ambiguity into the meaning attached to some inputs. Fortunately, a node in the refinement graph provides a well-defined context which helps reduce this ambiguity caused by merged classes. The most likely student inputs at a node are exactly those which correspond to transition phrases tagged to branches leaving that node, or nodes slightly lower in the graph. Therefore, if an input matches one of these branches, there is a high probability that the intended meanings are the same.

After several improvement iterations, this simple scheme allows the tutor to understand about 80% of student inputs using a vocabulary of about 1500 words.

6.2. Sorting lab and verifier

There are a number of reasons for exposing beginning programming students to the concepts of program correctness. In particular, the discipline of structured programming depends heavily on correctness proofs of program segments, and personal experience indicates inventing loop assertions for a program greatly increases the programmer's understanding of his routines. Unfortunately, few beginning programmers have the ability to carry out a correctness proof of their program, which suggests that a program verifier would be a valuable aid in teaching introductory programming.

Many of the verifiers which have been written, however, require intervention by the user to direct the activity of the verifier, which is not acceptable for beginning students. So it was decided to develop a program verifier which could verify simple programs without intervention from the student. To accomplish this, the particular domain of programs the verifier accepts was limited to programs for inplace sorting of the elements of a one-dimensional array. This domain was chosen for two reasons:

First, sorting programs are among the most used examples in introductory programming courses, and second, every program verifier constructed so far has verified several sorting programs, which provides a standard for comparing this verifier to previous work.

The verification system consists of three major components: an editor, an interpreter, and the program verifier. Let's consider each of these in a little more detail.

6.2.1. *The editor*

The editor allows programs to be written in a programming language with primitives especially designed for sorting (Figure 3). In-place sorting routines must conserve the keys they are sorting; hence the language provides two primitive operations for moving keys (*exchange* and *insert*), and does not allow assignment of values to the keys of the array. Successor and predecessor functions on the indices of the array, as well as a special *scan* statement, provide sequential access to the elements of the array. The language also includes *if*, *while*, and *call* statements. All procedures are allowed to be recursive.

The editor is designed to facilitate top-down program development. The program is internally represented as a tree; deletion or insertion of a subtree between any two nodes is permitted at any time. Also, the editor insists that the student complete each statement before inserting another (e.g., the *endwhile* of a *while* statement must be properly inserted before going on to other statements). Finally, since the language is sufficiently modest that nearly all its statements may be recognized by the first character, the editor completes program statements as soon as the statement type is recognized, allowing the programmer to concentrate on the program being developed.

The assertion language provides two predicates concerned with arrays, namely *sorted* (s,t) and *array*(s,t). Their meanings are *sorted* $(s,t) \Leftrightarrow$ if $s \leq r < j \leq t$ then $x(i) \leq x(j)$; *array* $(s,t) <$ *array* (u,v) $\Leftrightarrow$ if $s \leq i \leq t$ and $u \leq j \leq v$ then $x(i) < x(j)$. The language also contains predicates ($<$, $=$, $>$, $\leq$, $\geq$) for relating indices of the array.

```
< ptr > ← < ptr > ( ± 1 )

exchange < key > with < key >

insert < key > below < key >

while < boolexp > do

     endwhile

if < boolexp > then

               else

               endif

scan { up  } with < ptr > from < exp > to < exp >
     {down}


    endscan

procedure < name >

call < name >
```

Figure 3: Programming language statements

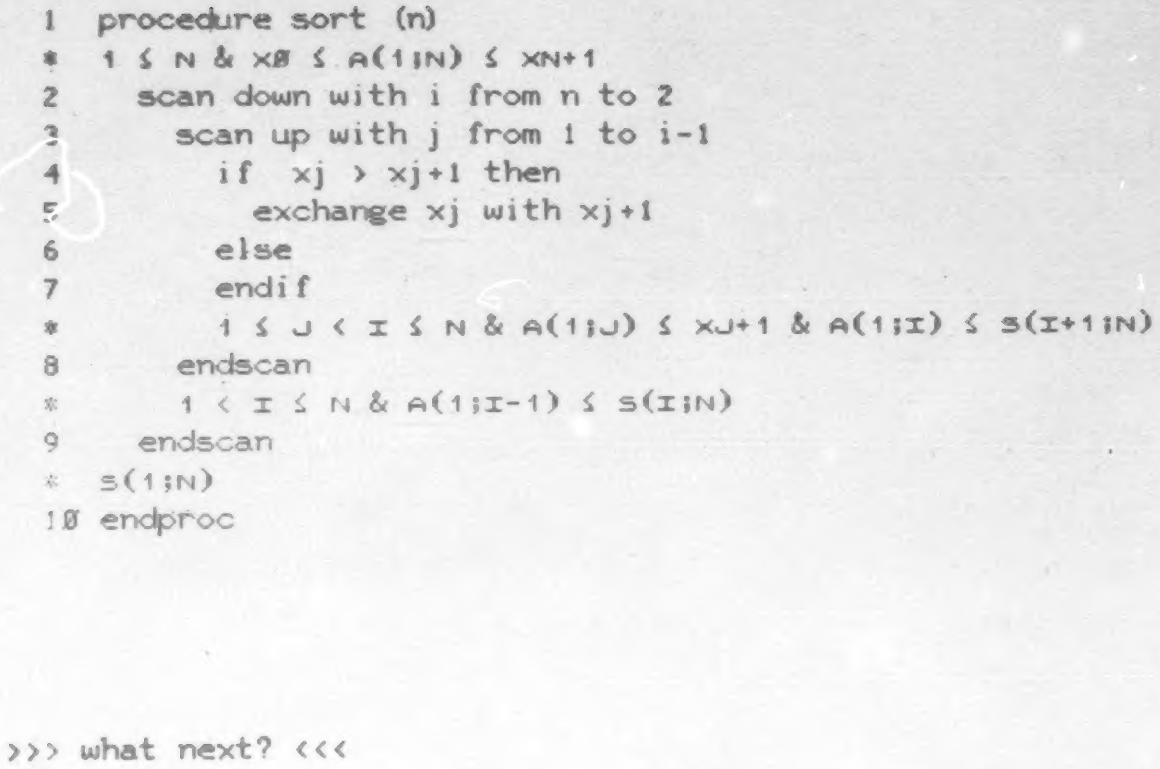
An assertion is then a sentence composed of these basic predicates and the connectives and and or. Notice that it is possible to express the negation of a pointer predicate in the language, but not an array predicate (i.e., sorted or array). The student is required to provide an assertion statement for each loop in the program: a loop body exit assertion (B_{EXIT}) and a loop exit assertion (L_{EXIT}). Examples of such assertions are in Figure 4.

6.2.2. The interpreter

The interpreter is capable of executing any program written in the programming language. During execution, the status of the array being sorted is dynamically displayed, along with the location of the various indexing pointers (Figure 5). Only the currently active procedure is displayed; as each new procedure is entered, that procedure is displayed along with the diagram of the array segment and a stack of procedure names giving an invocation trace. Both assertion language and programming language statements are executed, and the truth or falsity of the assertion language statements is indicated.

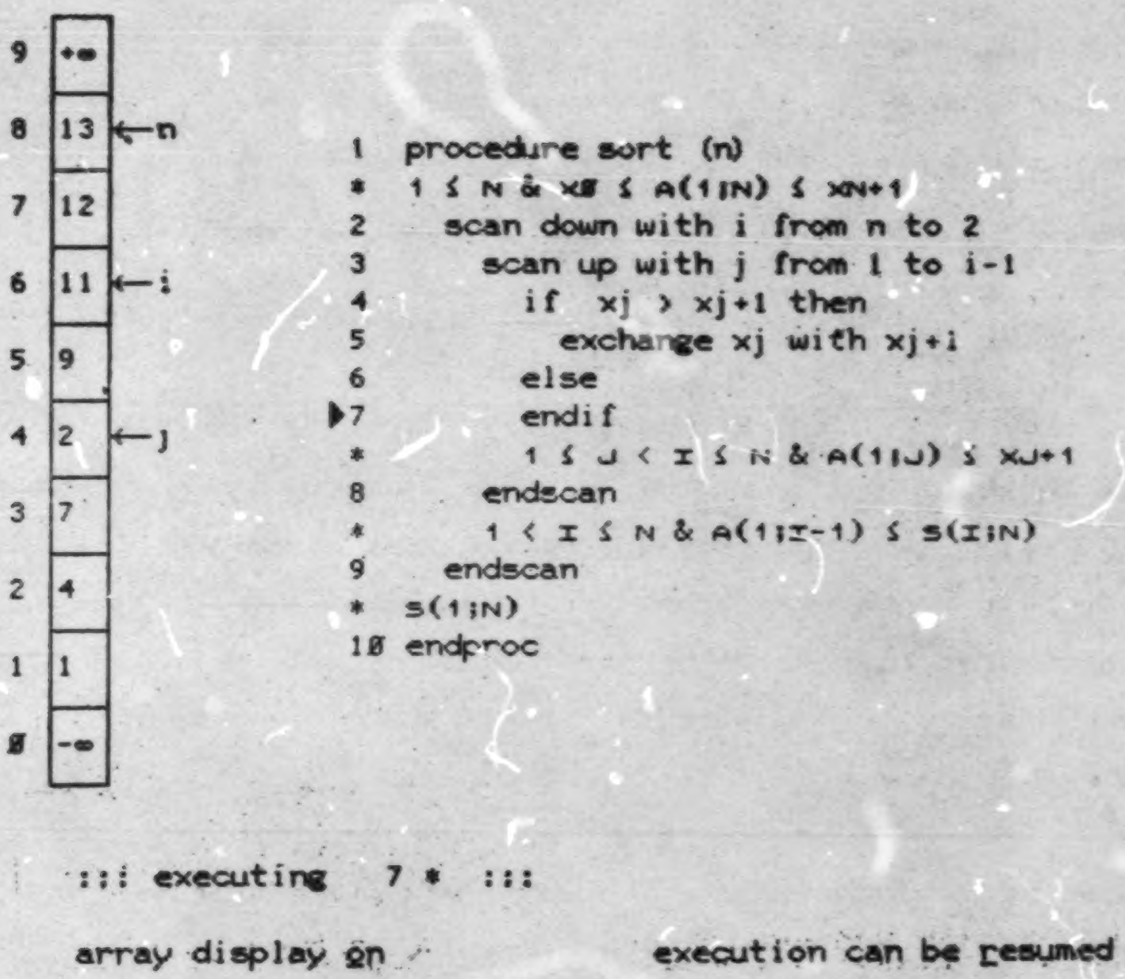
6.2.3. The verifier

Because this verifier is only concerned with a limited domain, it is faster than other program verifiers in existence. There are two reasons for this: first, since it is impossible for the program to destroy keys, the verifier only needs to prove the keys are sorted; second, because of the specific domain, the verifier has been designed to prove theorems which occur frequently very quickly, while perhaps taking longer than



```
1  procedure sort (n)
*  1 ≤ N & x0 ≤ A(1;N) ≤ xN+1
2    scan down with i from n to 2
3      scan up with j from 1 to i-1
4        if  xj > xj+1 then
5          exchange xj with xj+1
6        else
7        endif
*        1 ≤ J < I ≤ N & A(1;J) ≤ xJ+1 & A(1;I) ≤ S(I+1;N)
8      endscan
*      1 < I ≤ N & A(1;I-1) ≤ S(I;N)
9    endscan
*  S(1;N)
10 endproc



>>> what next? <<<
```

Figure 4: Sample sorting program



```
 1   procedure sort (n)
 *   1 ≤ N & x0 ≤ A(1;N) ≤ xN+1
 2     scan down with i from n to 2
 3       scan up with j from 1 to i-1
 4         if  xj > xj+1 then
 5           exchange xj with xj+1
 6         else
▶7         endif
 *         1 ≤ J < I ≤ N & A(1;J) ≤ xJ+1
 8       endscan
 *       1 < I ≤ N & A(1;I-1) ≤ S(I;N)
 9     endscan
 *   S(1;N)
 10  endproc

   ::: executing   7 *  :::

   array display on                  execution can be resumed
```

Figure 5: Sample execution display

usual to prove less typical theorems.

The verifier is composed of three distinct subsections. The first of these, the verification condition generator, is responsible for creating theorems to be proven from the student's assertions. For each loop in the program, two theorems are generated as follows. A loop body entry assertion (B_{ENTRY}) is generated from the body exit assertion (B_{EXIT}) by backward substitution. Now, assuming C stands for the loop condition, we must prove the two lemmas:

(1) B_{EXIT} and C implies B_{ENTRY}

(2) B_{EXIT} and $\neg$C implies L_{EXIT}(the loop exit assertion)

Beginning with the bottommost and innermost loop, and working outward, such lemmas are generated for the whole program. These are then passed to the second section of the verifier, the theorem prover, which proves or disproves all lemmas for the program. The third section is a counter example generator, which will provide the student with counter examples for any false lemmas.

Note that the theorem prover is the subsection which is specialized for sorting programs; the lemma generator is a completely general routine. Also note that the verifier will always terminate, and indicate whether the program is correct or incorrect with respect to the given assertions.

6.2.4. Performance of the system

The editor and interpreter alone provides a highly instructive sorting laboratory which has been well received by students who have tested portions of the system. The verifier alone, for those programs with which it has been tested, has proven to be the fastest system known. (It must be noted, however, that other verifiers can handle relatively arbitrary programs, while ours is limited in its domain). A typical bubble sort routine, for example, requires nine CPU-seconds to verify. Unfortunately, this may require as long as 30 clock-minutes during periods of heavy system load, which severely hampers its usefulness for instruction.

6.3. Program anomalies detectable by an automated system

Beginning programming students learning their first programming language normally have a very "narrow" view of the problem solving process. They learn the function of each of the individual statements in the particular language but are not familiar with all the language features and lack the insight to select the most appropriate language constructs for a particular problem they must solve.

Among some of the reasons why this occurs are:

- Lack of experience,
- Teaching technique,
- Lack of desire to expand their own programming ability (usually caused by lack of interest), and
- Misunderstandings or misconceptions.

Because students:

- Start coding before they understand all aspects of the problem.
- Program piecemeal and add "fixes" to patch up incomplete algorithms instead of restructuring or changing the basic algorithm, and
- View their program as a series of essentially unconnected statements without reflecting on more global aspects of their program

"Poorly structured" programs are often produced. Here, "poorly structured" refers not only to control flow but also to inefficient, ineffective, or erroneous data flow.

An automated system capable of performing the global flow analysis that the student fails to do is clearly appropriate. Such a system capable of:

- Detecting program anomalies,
- Giving detailed information about the anomalies, i.e., helping the student understand what is wrong, and
- Helping direct the student in correcting the anomalies would be a valuable pedagogical tool.

6.3.1. Data collection

A set of four machine problems given as assignments in a beginning programming course of approximately 60 students has been collected. The course used Fortran as an implementation language and was directed toward Engineering students. The final solutions (those handed in for grading) are currently being hand analyzed for program "defects" dealing with:

- Programming style,
- Efficiency, and
- Language and algorithm misconceptions.

A report presenting:

- A categorization of these "defects",
- Reasons why students produce such "defects",
- Statistics on "defect" frequently, and
- Which "defects" are automatically detectable

will be completed within a few months.

6.3.2. Techniques

The thesis involves the use of global flow analysis techniques (both currently existing and newly developed) to detect anomalies in programs.

A flow graph corresponding to the student's source program is produced. This flow graph is then used by iterative techniques similar to those developed by Kildal ["A unified approach to global program optimization", SIGACT SIPLAN pp. 194-206, Oct. 1973] to perform each of the specific global flow analyses.

A uniform iterative global flow framework has been developed which encompasses most of the "standard" (i.e., "Live" variables, common subexpression elimination, dominance, etc.) and newly developed (i.e., unreferenced data, unititialized variable, transfer variable, etc.) analyses. Since these techniques do not involve "interval" analyses, the underlying program flow graph need not be reducible.

6.3.3. Specific program anomalies

This section presents examples of some of the anomalies to be detected. Each can be detected by the techniques mentioned in section 3 without any knowledge of the user algorithm being implemented.

Figure 6 is a subroutine implementing the binary chop method of root finding and will be used to present specific examples of anomalies to be detected. This is the type of code many beginning Fortran programmers produce as a final product (i.e., turned in to be graded).

```
      SUBROUTINE BINCHP(XL,XR,EPS,DELTA,ROOT)
      YL = F(XL)
      YR = F(XR)
      IF(YL*YR.GT.0) GOTO 10
   20 ITER = 0
      IF(ABS(XR-XL).LE.EPS) GOTO 30
      XM = (XL+XR)/2.
      YM = F(XM)
      ITER = ITER + 1
      DELTA = ABS(XR-XL)/2.
      PRINT,ITER,XM,DELTA
      IF(YL*YM.LT.0.) GOTO 40
      XL = XM
      YL = YM
      GOTO 20
   40 XR = XM
      YR = YM
      GOTO 20
   30 ROOT = XM
   10 RETURN
      END
```

Figure 6: Binary chop routine

6.3.3.1. Unreferenced data

Unreferenced data occurs when:

- A value, D, is assigned to a variable, V, and
- That value is not referenced by any statement of the program.

This can happen in a combination of 2 ways:

1) Variable V is assigned a new value prior to a reference, or
2) The variable V is never referenced, i.e., an "exit" is encountered prior to a reference.

Example:

At L17 of Figure 6, a specific value is assigned to 'YR'. However, there is no ancestor of L17 which references 'YR'.

6.3.3.2. Uninitialized variable

A variable, V, referenced at a specific statement, S, may be:

- Totally uninitialized

  i.e., no execution path from the beginning of the program to S assigns a value of V, or
- Partially ininitialized

  i.e., there is at least one execution path from the beginning of the program to S which does not assign a value of V.

Example of partially uninitialized variable:

Consider 'XM' referenced at L19. Assuming 'XL' and 'XR' are sufficiently close upon entry to the subroutine, i.e., ABS(XR-XL) <=EPS, then the flow of control might be (L1,L2, L3,L4,L5,L6,L19,L20). This execution path leaves 'XM' uninitialized when referenced at L19 and, thus, an erroneous root is returned.

6.3.3.3. Code motion

Code motion can be suggested as a correction to certain anomalies when the student asks for help. For instance, assume the partially uninitialized 'XM' at L19 has been detected. The suggestion to move L7 between L5 and L6 can be automatically generated.

6.3.3.4. Transfer variable

A variable is a transfer variable if:

- The value of an expression X is assigned to V, and
- At each reference (normally only one) to V which contains the value of X, the defining components of X have the same value as when X was assigned to V.

The reason for detecting such a situation is that the assignment of X to V can be eliminated and the expression X substituted for corresponding references to V. Although such a substitution probably produces a more efficient program, this is not the major reason for bringing this to the student's attention. The primary motivation is to help the student understand how data flows through his program.

Examples:

1) 'F(XR)' assigned to 'YR' at L3 can be substituted for 'YR' at L4 (thus, eliminating L3).
2) 'XM' assigned to 'ROOT' at L19 can be substituted for 'ROOT' at L1. This eliminates L19 and since no explicit action must be performed before returning, the 'GOTO 30' at L6 can be replaced by 'RETURN'.

There are several situations which, even though detected, should not be presented to the student. Two such situations are:

1) The transfer variable is assigned the value of an expression requiring computation (i.e., not just the value of another variable) outside a loop but is referenced inside the loop. Clearly, the value of the expression is invariant to the loop and its computation has been placed outside the loop for execution efficiency.
2) The transfer variable is assigned the value of an expression requiring computation and is referenced more than once. Thus the computation would have to be performed more than once if a substitution were done.

Example:

'F(XM)' assigned to 'YM' at L8 can be substituted for 'YM' at L12, L14 and L17. However, this produces two functional evaluations each time through the loop when only one is needed.

6.3.3.5. Initialization inside loop

When building an "IF" loop, the beginning student often places the initialization of the loop inside the loop. The two concepts of code motion and transfer variable can be used as a partial solution to detect and correct this situation.

Example:

As the subroutine is currently structured, 'ITER' at L5 is a transfer variable (i.e., the '0' assigned to 'ITER'

> at L5 can be substituted for 'ITER' at L9). If L5 is moved out of the loop, say to L4.5, 'ITER' is no longer a transfer variable because the data referenced through 'ITER' at L9 now has two sources.

Thus, if movement of the assignment to a transfer variable to a position outside the loop changes its status, it is a candidate for a misplaced initialization.

6.3.3.6. Common expression detection

Students often calculate expressions with exactly the same value several places in their program. Such duplications can be automatically detected. The purpose of bringing this to the student's attention is not to produce more efficient code (since an optimizing compiler will eliminate such redundant computations) but to help the student better understand how information flows through his program.

Example:

> The value of 'ABS(XR-XL)' computed at L6 is exactly the same as that computed at L10. A temporary variable can be used to transfer this value to the two places it is used.

6.3.3.7. 'GO'ing to a 'GOTO'

The 'GOTO' is standard tool used (especially in languages like Fortran) to handle momentarily unresolved actions. When these actions are finally resolved, the student fails to perform simple optimizations in order to simplify the control structure and produce a more understandable program. A class of such "defects" is the explicit transfer of control to an unconditional transfer of control.

Example:

The 'GOTO 10' at L4 can be replace by 'RETURN'. Such a form is more easily understood by someone reading the program and better reflects the intended meaning.

6.3.3.8. Local variable in a parameter list

Students will often place a local variable of the subroutine in the parameter list. This can often be automatically detected even if the corresponding argument is actually manipulated in the calling routine (although computations involving the argument are normally completely absent).

Example:

The variable 'DELTA' in the parameter list at L1 is probably a local variable. Since 'DELTA' is assigned prior to any reference, it cannot be an input variable. Assuming the value returned to the calling routine is never referenced (see section 6.3.3.1.), it cannot be an output variable, and it can be concluded that 'DELTA' is a local variable.

6.3.3.9. Modification of input parameter

It is generally considered a poor programming practice to modify an input parameter in a subroutine (of course, a parameter may be used for both input and output). Such a practice can cause erroneous results (if the corresponding argument is referenced expecting it to have its original value) or excess computations to recalculate the original value of the argument.

Example:

> 'XL' and 'XR' in the parameter list at L1 are clearly input parameters since they are referenced before they are assigned. If the values returned to the calling routine are referenced, the programmer may incorrectly assume he is referencing the original input values. If the returned values are never referenced (i.e., the parameters are not output parameters) program anomalies may occur when the subroutine is used in a different environment.

References

[1] Danielson, R. and Nievergelt, J. (1975a). An automatic tutor for introductory programming students. Proc. Fifth Symp. on Computer Science Education, SIGCSE Bulletin, Vol. 7, No. 1, February 1975.

[2] Danielson, R. L. (1975b). PATTIE: An automated tutor for top-down programming. Report UIUCDCS-R-75-753 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, October 1975.

[3] Gillett, W. D. (1976). An iterative program advising system. Proc. of SIGCSE-SIGCUE Joint Symp. on Computer Science Education, SIGCSE Bulletin, Vol. 8, No. 1, February 1976.

[4] Gillett, W. D. (1977). Iterative techniques for detection of program anomalies. Submitted to the Conf. on Principles of Programming Languages, Los Angeles, California, January 1977.

[5] Mateti, P. (1976). An automatic verifier for a class of sorting programs. (Ph.D. Thesis), to appear as DCS Report, September 1976.

7. Experimental and formal language design applied to control constructs for interactive computing (D. W. Embley)

7.1. Introduction

This research [Embley, 1976b] explores two objective approaches to language design and applies them to an investigation of control constructs for interactive computing, particularly in Computer-Aided Instruction (CAI). In an experimental approach to language design, a designer recognizes the human element in programming and attempts to achieve an optimal design by an empirical investigation of language constructs. Through carefully documented, thorough, and replicable experiments, designers can present objective evidence to support claims about language features and stylistic considerations. In a formal approach to language design, a designer recognizes the theoretical foundations of programming languages and attempts to achieve an optimal design by a specification of the properties of language constructs in order to expose weaknesses, inconsistencies, and design flaws. A better understanding of the syntax and semantics of language constructs can make it easier for a language designer to objectively see what features are really desirable.

These approaches to language design are not only applicable in an investigation of proposed language constructs but also in an investigation of language design principles. Many lists of design principles exist, but notions such as "simplicity" or "uniformity" that are generally included in these lists are insufficiently defined. A formal definition of these principles would facilitate identification and consideration of language features that violate basic design principles. Moreover, experiments can be applied in an attempt to objectively validate design principles so that language designers can confidently apply them.

7.2. Approach

As a means to explore experimental and formal language design, D. W. Embley has designed a new programming language called KAIL [Embley, 1975a]. KAIL was originally motivated by a desire to improve TUTOR [Sherwood, 1974], the author language for the PLATO IV CAI system [Alpert and Bitzer, 1970]. In KAIL, a selector construct [Embley and Hansen, 1976] is introduced to handle CAI answer judging; this construct also unifies selection and iteration and subsumes most typical high level language constructs (e.g., if-then-else, while, repeat-until). Figure 1 gives the essential syntax and semantics of the KAIL selector. A static exception processing scheme is also introduced to handle frame sequencing.

These KAIL control constructs were tested in two experiments conducted on-line in a CAI environment, and the results indicate that they are likely to be psychologically sound. In an experiment on the KAIL selector [Embley, 1975b], subjects attempted to understand and answer questions about two short programs. For one group of subjects, these programs were written in S, a language containing the selector; and for the other group, they were written in A, an ALGOL-like language. In the other experiment on frame sequencing [Embley, 1976a], subjects debugged and modified a substantial CAI lesson about 500 lines in length. One version, T, was written in a TUTOR-like language, and the other version, K, was written in a KAIL-like language.

These KAIL control constructs were also examined through a formal definition of their semantics, and their properties were clearly exposed.

syntax:

```
    selector → [statement-sequence  control  choices]
             → [statement-sequence  loop-control  choices]
     control → if expression
loop-control → while expression
             → until expression
     choices → | relation expression : statement-sequence
             → | relation expression : statement-sequence  choices
```

semantics:

The selector is executed in 5 steps

1. Execute the initial statement-sequence.
2. Evaluate the control-expression and save its value in a temporary, t.
3. Test each choice in turn until a "selected choice" is found such that

   (t relation choice-expression)

   yields <u>true</u>.
4. Execute the statement sequence in the selected choice.
5. Execution now continues as follows: For <u>if</u>, exit. For <u>while</u>, if no selected choice is found, exit; otherwise, return to step 1. For <u>until</u>, if no selected choice is found, return to step 1; otherwise, exit.

Figure 1: Essentials of the KAIL selector

An axiomatic approach was applied to the KAIL selector, and the KAIL exception processing scheme was defined in terms of a behavioral model. The TUTOR exception processing scheme was also formally defined and compared with the KAIL scheme.

7.3. Conclusions

7.3.1. Design principles

As a result of the investigation of experimental and formal approaches to language design, three basic design principles evolved:

1. Uniformity,
2. Separability, and
3. Locality.

These three principles are proposed as a possible basis for an informal approach to language design.

The uniformity principle suggests that languages ought to be designed with a one-to-one relationship between syntax and semantics. In a program written in a uniform language, a single semantic notion consistently has the same syntactic form. Moreover, a single rule applies to each language construct independent of its context. In a nonuniform language, there are several ways to express a single semantic notion or various possible meanings for a syntactic form depending on the execution history, so a programmer has more to consider. Nonuniformity leads to more decisions and thus more probability of error.

The separability principle suggests that special purpose composite structures may be harmful. The advantage of a composite construct lies in its power to produce a desired effect with a minimal amount of code. Most high level language structures are composite constructs; and so long as no programmer needs an unavailable component of a composite construct, all is well. In the unfortunate situation where a needed component is unavailable, the language designer may be willing to extend his language. If not, a programmer would have to obtain the component indirectly by "programming around" the problem, if this is possible; otherwise he would have to use a different language. To supply syntactic forms for many composite structures would cause language constructs to proliferate. To force the programmer to indirectly separate composite structures, on the other hand, would result in code that is more difficult to understand and maintain. Separability suggests that a few general language constructs are better than many constructs that have only specific utility and that caution should be exercised in the creation of special purpose constructs.

The locality principle suggests that language features should be as permanent and local as possible. Locality aids programmers because it structures information. When a programmer has a large amount of information to consider, any mechanism that structures this information or restricts it so only a small subset needs consideration is most helpful. When a programming language encourages locality, it reduces the amount of text a programmer must consider in order to determine the effect of a language construct. Furthermore, it imposes a structure on information accessing methods and restricts the variability of language features whenever possible. Locality also facilitates modularity and is particularly valuable when a program must be modified.

In [Embley, 1976b] these principles are formally defined. Moreover, the experiments conducted in this research lend support to these principles, particularly the sequencing experiment.

7.3.2. The experimental approach to language design

The results of the selector experiment support the hypothesis that programmers understand the KAIL selector more easily than an equivalent set of traditional constructs. S language subjects answered more questions correctly than A language subjects and also thought they initially better understood the S-favored program. Statistics on the number of questions initially answered correctly, average time taken to obtain a correct answer, and initial and final self-evaluations are all in the direction of the S language. No performance statistics favor the A language.

In the sequencing experiment, the results generally lend support to all three basic design principles. In T, the behavior of a procedure is context dependent, but in K, the behavior is independent of context. T subjects introduced errors due to this context dependency when they improperly inserted new procedures. This lends support to the uniformity principle. Several observations support the separability principle. T subjects had difficulty separating composite constructs, and in one instance, none of them were able to find a way to "program around" a particular problem. The experiment also supports the locality principle because T subjects consistently failed to reset global status information when they attempted to fix one of the bugs. In K, this information was local and caused no particular problem.

The experimental approach to language design can produce scientific evidence to support claims about language features and design issues as shown in both experiments. Through empirical tests, designers can gain assurance that their language features are psychologically sound, and they can gain confidence in language design principles.

7.3.3. Experiments in the PLATO environment

Since these experiments were conducted on PLATO, they also illustrate the applicability of an on-line methodology for conducting experiments in programming. Several advantages can be gained by conducting experiments on-line in an interactive, CAI environment. There can also be several disadvantages.

The advantages include:

1. A controlled teaching environment,
2. The ability to interact meaningfully with subjects during an experiment,
3. Individualized sequencing,
4. The ability to gather highly precise data,
5. The ability to impose strict timing constraints,
6. Assistance in grading,
7. On-line editing capabilities, and
8. On-line execution capabilities.

The disadvantages include:

1. Cost,
2. System failures, and
3. Subject unfamiliarity with the system.

In general, the experiments profitably took advantage of the PLATO environment. Time and space limitations, however, prevented full exploitation of potential advantages.

7.3.4. The formal approach to language design

An application of an axiomatic formalism to the KAIL selector helped clarify and concisely specify several general observations. The KAIL if and case are semantically identical, and all is complex compared to the other control types. The formalism also revealed similarities and differences among if, while, and until, and led to an investigation of another possible control type.

The semantics of both KAIL and TUTOR exception handling were defined in terms of a special purpose abstract machine. This behavioral definition shows that the KAIL constructs adhere to the locality principle better than the TUTOR constructs. Moreover, the formalism shows how the TUTOR constructs violate both the uniformity and separability principles.

The formal approach to language design can objectively reveal properties of language features as shown in the formal definition of the KAIL control constructs. It can also objectively expose weaknesses and inconsistencies and provide insight into why some language features are better than others.

7.4. Summary

The results indicate that further research in experimental and formal language design is likely to be fruitful. These methods can be applied to obtain objective evidence to support claims about language features and design issues in general. It would be particularly valuable to further apply these methods to obtain additional support for the three basic design principles. These could then be confidently used as a partial basis for reasoning about and designing programming language features in general.

References

[1] Alpert, D. and Bitzer, D. L. Advances in Computer based education. Science, 167, (20 March 1970), 1582-1590.

[2] Embley, D. W. An experiment on a unified control construct. Technical Report No. UIUCDCS-R-75-759, University of Illinois at Urbana-Champaign, Department of Computer Science, August 1975b.

[3] Embley, D. W. An experiment on CAI sequencing constructs. Technical Report No. UIUCDCS-R-76-771, University of Illinois at Urbana-Champaign, Department of Computer Science, February 1976a.

[4] Embley, D. W. An introduction to KAIL. Lesson kaids on the PLATO System, University of Illinois at Urbana-Champaign, August 1975a.

[5] Embley, D. W. Experimental and formal language design applied to control constructs in interactive computing. Technical Report No. UIUCDCS-R-76-811, University of Illinois at Urbana-Champaign, Department of Computer Science, July 1976b.

[6] Embley, D. W. and Hansen, W. J. The KAIL selector - a unified control construct. SIGPLAN Notices, Vol. 11, No. 1, January 1976, 22-29.

[7] Sherwood, B. A. The TUTOR language. Computer-based Education Research Laboratory and Department of Physics, University of Illinois at Urbana-Champaign, June 1974.

3. Use of ACSES in instruction (R. G. Montanelli, Jr., E. R. Steinberg)

An introductory computer science course at the University of Illinois ordinarily consists of 2 large lectures taught by a professor and a smaller discussion taught by a teaching assistant (TA) each week. The lectures introduce most new material, while the discussions are small classes in which TAs answer questions and help students with their computer programs. Since somewhat more than one-half of the lecture time was typically spent on FORTRAN, it was initially decided to develop a sequence of PLATO lessons to teach FORTRAN and to use these lessons to replace one lecture a week, throughout the semester.

Work on the lessons was begun in the fall of 1973 by students in an honors course. During the course of the project, most lessons were written by students. In addition to being in computer science, many of the students had some teaching (or teaching related) experience as teaching assistants, consultants, and graders. All the lessons went through numerous stages of testing followed by revisions, corrections, and improvements. Ultimately lessons were polished by highly experienced staff members or students.[1] The lessons were not designed according to some particular theory of instruction because it is not clear that a suitable one exists (Anastasio, 1974). It was felt that the best way to arrive at a good set of lessons would be to encourage varying styles and techniques, and to ultimately choose the best lessons on the basis of their effectiveness or on student preferences. Of course this could

---

[1] The authors would especially like to thank Professor H. G. Friedman, Jr., Sandra Leach, and Jeffrey Barber for their invaluable help in this area.

(and did) result in some lessons which had to be completely rewritten, but it avoided the possible trap of using a theory which might not apply. More details about the development of the lessons are given in Montanelli (1975), while Barber (1975) reports an indepth study of the design, evaluation, and subsequent revision on the basis of data collected during use, of one lesson.

8.1. Fall, 1974

The initial use of 12 of the lessons to replace classroom instruction occurred in the fall of 1974, although optional, voluntary use had occurred for some lessons during the previous spring and summer. A relatively small class (50 students) taught by the first author was selected for the first actual test. In order to obtain some comparisons between the lessons and ordinary classroom instruction, the class was randomly divided in half, with one-half receiving the traditional two lectures and the other having a PLATO lesson replace a lecture each week. There were three interesting results from this early experiment. First, questionnaires administered at varying points during the semester indicated that although students seemed reasonably satisfied with PLATO early in the semester, the comments made at the semester's end indicated some dissatisfaction. Four possible explanations were: 1) early in the semester students found PLATO new and interesting, but the novelty wore off by the semester's end, 2) students were more concerned with grades by the end of the semester, 3) earlier lessons were in better shape than later ones, and 4) a computer memory (ECS) shortage made it impossible for a group of students in a room of PLATO terminals to use many different lessons simultaneously.

Undoubtably each of these explanations played some part in student attitudes, and of course little could be done to change the effects of the first two. However, it was the case that some of the earlier lessons had been better tested than some of the later ones. Thus some additional improvements for these lessons were indicated. Also, the shortage of ECS could have had a larger effect at the end of the semester, because as students fell behind or needed to review, they created a demand for many lessons at the same time, and that wasn't possible.

The most encouraging result was a correlation of .58 between the amount of time spent in the required PLATO lessons and the course grade. If this was a cause and effect relationship, it illustrated the usefulness of the lessons.

The final interesting result was the lack of a significant difference between the two groups on achievement variables. With some of the problems encountered, this result was good, in spite of the fact that all observed differences favored the non-PLATO group, with two exams almost showing significant differences at the .05 level. Other results showed similar, low drop rates in both groups, and no relevant differences from the previous year's class on the results of a student rating of instruction form. More detailed results appear in Montanelli (1975).

Three possible methods for improving the instruction gotten through using the PLATO FORTRAN lessons were identified. They were: 1) increase ECS; 2) revise lessons, especially to include more exercises and other interaction; and 3) consider more strongly encouraging students to use the PLATO materials. On line data indicated that on the average, students probably did less than one-half of the assigned material. A computer-managed instruction (CMI) program (Anderson et al, 1974) had already shown that PLATO could be used successfully to increase student performance through CMI only. A later experiment in fall 1975, considered this question. Also, ECS was added in January of 1975, solving one problem, and revisions of lessons were begun along lines indicated.

The study concluded that CAI materials initially written by students could replace some lectures on the FORTRAN language in an introductory computer science course. However, it was obvious that more effort must be spent on lesson development and evaluation than was originally suspected.

8.2. Spring, 1975

The purposes of this evaluation were fourfold: (1) to get baseline information on students' attitudes and to determine if there were changes during the semester; (2) to assess the data collection in CS records; (3) to provide guidelines for revision and improvement of some of the lessons; and (4) to provide recommendations for improved implementation and integration of PLATO into CS courses.

8.2.1. Students' attitudes

It is particularly important to evaluate student attitudes when a new technology is introduced. A positive attitude is a necessary though not always sufficient condition for learning. The student's attitude must be sufficiently positive that she/he is willing to try the CAI lessons. Student attitudes can also serve as a valuable resource of information for revising and improving both the lessons themselves and methods and procedures in course implementation. Although PLATO had been widely used in a variety of courses at the University of Illinois, it was anticipated that most of these beginning CS students would not have had prior experience with it. What was their initial reaction to the prospect of using PLATO and on what basis was this made? As expected most of the students[2] in the sample (71 out of 99) had not had previous PLATO experience. Responses to an open ended question revealed that about 39% of the initial reactions were positive, 44% indicated fear or displeasure, and 24% could not be interpreted positive or negative.

Thus, although most of the students had not had previous experience on PLATO, their expectations were not negative. For the most part, they were uncertain or favorably disposed. Their comments revealed some concern and some confusion, not knowing what to expect. Their sources of information about PLATO were mainly other students who had courses on PLATO and instructors in this course. This information may not have been entirely relevant because other courses may have used PLATO in different ways than CS 105, e.g., supplementary drills or computer management of instruction. It seems, therefore, that students need some specific information about CS 105 and PLATO.

Three questions were asked at both the beginning and end of the semester to assess possible changes in attitude toward PLATO per se. Students were asked to rate each statement on a 5-point scale, from strongly agree to strongly disagree. There was no statistically significant difference between scores comparing the entire initial sample of 99 to the end of semester sample of 75. Nor was there a significant difference when the data were limited to the 56 students who filled out questionnaires both in January and April. However, as can be seen from Table 1, the proportions of students who were uncertain in January decreased by April. The shift was toward disagreeing that PLATO is an "expensive gimmick", disagreeing that it is dehumanizing, but agreeing that PLATO was enjoyable.

---

[2] Questionnaires were handed out randomly to students in different PLATO sections.

Fifty-seven percent of the students saw "better understanding of the material" as the most important advantage of PLATO. More than half of the students (54%) indicated they would advise a friend to take a PLATO U. of I. course if at all possible, 30% only if convenient, and 16% to avoid it. The mean response to this question was not significantly different from the mean of students surveyed in a 1974 study of 584 students in 23 courses (Siegel, 1974). Although 69% of the students felt that the PLATO presentation was most effective for the material covered, 30% felt the presentation would have been equally or more effective by lecture and/or textbook. A few students commented that they favored PLATO because the lectures were "worthless." In other words, it wasn't that PLATO was so great, but that some of the lectures were so poor.

What "bugs" students most about PLATO is when they come a "long" distance and the system is down. Unfortunately, during the first 10 days of the semester the system was down most of the time.

With respect to the lessons, students are most irritated by inadequate response judging. The lesson most frequently cited as bothersome was fortcomp: response time was too slow and it was

Table 1

Attitudes of CS 105 Students to PLATO at Beginning and End of Spring, 1975 Semester

	AGREE OR STRONGLY AGREE		DISAGREE OR STRONGLY DISAGREE		UNCERTAIN	
	JAN.*	APR.**	JAN.*	APR.**	JAN.*	APR.**
[1]Expensive gimmick	.04	.07	.73	.81	.27	.12
[2]Dehumanizing	.16	.17	.58	.69	.26	.13
[3]Enjoy PLATO	.48	.65	.10	.19	.42	.16

1 Computer-based education is nothing but an expensive gimmick.

2 Computer-based education dehumanizes the student.

3 I enjoyed (or expect to enjoy) using PLATO this semester.

* N = 99

** N = 75

cumbersome to cope with. Another major complaint was lack of clarity, such as poor directions, inadequate explanations, lesson was vague or confusing. Some complained that lessons were too long or boring.

Special questionnaires were distributed for the lessons on FORTRAN arithmetic and FORTRAN formatting. Mean overall rating for the FORTRAN arithmetic lesson was 3.9, on a 5 point scale where 4 was "good". The advantages cited for doing this lesson on PLATO were: feedback, motivation (fun), active participation and seemingly easier to learn. The students who did the lesson on PLATO preferred that medium of instruction over a paper handout of the lesson. After taking the format lesson, students indicated that what they liked most was being able to work at their own pace and that exercises and practice were available. Fifty-six percent of the students indicated they preferred PLATO to a lecture on this material, 27% preferred a lecture, and 12% said they had no preference.

8.2.2. Data collection

The data collection in CS records kept track of the number of times a student had signed in to a lesson and the total time spent on a lesson. Data did not tell whether a student had completed the lesson and repeated it or had done part of the lesson each time he signed in. Data revealed that the average time spent on some of the lessons in class tends to exceed the allotted class time. Some examples are given here.

lesson	mean time	S.D.	range	mean number of times in lesson
fortif	48.63	17.4	18-82	2.37
loops	61.46	19.6	30-116	2.11
fortarrayl	60.79	22.9	35-144	1.68

Another useful aspect of the data collection was the record of last date the students had signed on. A quick glance enabled the instructor to check up on attendance. In April a random sample of 26 students in each of the six CS 105 sections was chosen. Only 60% of these 156 students had signed onto the system within the preceding 2 weeks. The reasons were not ascertained in this case.

8.2.3. Lessons, instructional design and learning

It seemed that some of the lessons did not make much use of the interactive feature of PLATO. Lessons varied in the number of exercises provided, requirement of a specific criterion of performance in order to proceed, and frequency of sets of exercises.

A study was designed using lesson fortarith to investigate the effects of learner control, placement of exercise sets, and PLATO versus varian copy handouts of the lesson. Unfortunately, the PLATO system was down most of the time during which the experiment was to have taken place. It was also down during the preceding weekend when the teaching assistant was to have set up implementation, and she was unable to check out her work.

One should not assume results of the 10-minute quizzes were dependable for drawing conclusions. However, they might provide some tentative insights.

Mean Scores on a 10-point Quiz Given by TAs in Quiz Sections

	Coercive	Non-Coercive	Handouts
Frequent sets of exercises	7.76	7.26	6.01
Few sets of exercises	7.06	6.84	7.51

It is difficult to understand why students with fewer exercises on handouts did better than those who had more exercises. (The reverse was true on PLATO.) Note that the highest scores were obtained by those who had more frequent sets of exercises and who were in coercive conditions. It was decided to repeat the experiment again the next semester, when it was anticipated that the system would be more stable and the lesson thoroughly tested. Results are summarized in section 8.3.

A second experiment was set up for lesson fortfmtl. The purpose was to see which instructional conditions are most facilitative. The factors in the 2 x 2 x 2 design were (1) coerciveness (required or optional), (2) instructions to do problems or do them correctly, and (3) size of exercise set (2 or 4 of each type). Do students follow suggestions in the instructions about how much practice they should get? Table 2 shows that students in the optional conditions did more I-and F-format problems then in the required conditions. (This may have been due to an artifact of the lesson. Required students were not given the opportunity to do more than required.) But in the E-formats optional students did fewer problems. The same pattern emerges when students are told how many problems to do correctly (Table 3). In the I- and F- formats optional did more than suggested, in E- format they did fewer. This may have been related to problem difficulty. Table 4 shows that all students did a rather high percentage of problems correctly in I- and F-format exercises. However, in the E-format exercises, the percentage of problems done correctly was only 51% in required conditions and 44% in optional. It is apparently the case that when the problems are not too difficult, students follow suggestions

Table 2

Mean Number of Problems Done by Type of Exercise and Experimental Condition

	Small Sets Given		Large Sets Given		
	Do	Do Right	Do	Do Right	Total
I-format (N=316)					
Required	4.2	5.4	8.2	9.1	6.7
Optional	8.2	8.4	12.4	12.2	10.3
F-format (N=300)					
Required	6.5	7.0	12.0	13.8	9.8
Optional	9.7	9.4	13.5	14.4	11.7
E-format (N=242)					
Required	6.0	11.1	12.0	18.7	11.9
Optional	8.1	8.8	11.8	12.8	10.4

Table 3

Number of Problems Done Correctly

	Instructions	
	Do	Do Right
I-format		
Required	5.2	6.1
Optional	8.2	7.9
F-format		
Required	7.8	8.9
Optional	9.1	9.1
E-format		
Required	4.0	8.6
Optional	5.0	5.5

Table 4

Percent of Problems Done Correctly

	FORMAT		
Condition	I	F	E
Required	84.2	86.1	50.8
Optional	76.0	77.6	43.6

for how much to do, or practice even more. But when some difficulty is encountered, they do significantly less than required. It should be pointed out that optional students did a lower percent correct and this was statistically significant. However, it might not be educationally significant. That is, a student performing at 76% accuracy might not do any worse on a subsequent achievement test than a student operating at an 85% accuracy level. There may be considerable difference between students operating at 51% and 44% levels. In fact, neither of these levels would seem to be satisfactory in terms of adequate understanding.

Apparently a reasonable minimum requirement for practice is not abrasive to students. Although not essential for less difficult concepts, it seems necessary for more difficult ones. Also, the difficult problems (E-format here) should be accompanied by some form of corrective information in feedback and/or help sequences.

8.2.4. Classroom observations (course implementation)

Students took notes on lessons. A questionnaire was distributed after students had completed lesson fortif. Results showed that about 2/3 of the students took notes on lessons fortif and fortarith and that 3/4 of the students who took notes knew that the material was covered in the text.

Whenever the classroom was visited, a proctor was seated at a terminal near the door. She/he wore no identification nor was there any sign on the terminal. Students had no way of knowing that a "human being" was available for help. Furthermore, proctors generally were busy programming or playing a game on PLATO, so that if a student did raise a hand for help, it was unlikely to be acknowledged.

8.2.5. Recommendations

1. More extensive and better communication should be established between instructors and students as well as between proctors and students. CS 105 instructors might help create a more positive attitude by orienting students as to the goals of the PLATO lessons in CS 105: how much time they will take, what they can expect from the lessons, who will help them if there are problems, and so on. It is also important that they tell the students that PLATO is used differently by different courses; therefore, previous negative experience may not be at all applicable here.

Provide a large sign for the proctor to put on top of his terminal so that students know that a person is in the room and available. Proctors should be oriented to the responsibility of walking around the classroom periodically. Also, they should have worked through each of the lessons themselves.

The attendance, or lack of it, at PLATO sessions should be investigated. It may be part of an overall student syndrome of generally poor class attendance at this time in the semester. It may be in part a reflection of student attitude toward the usefulness of the PLATO lessons at this point in the course.

2. Revise data collection to include the time it takes to complete each lesson, number of times the student has completed entire lesson, and number of times the student has entered lesson.

3. Lesson revisions should be aimed at minimizing student frustration and using more of PLATO's interactive capability. The length of time it takes students to complete a lesson must be compatible with amount of time allotted. Some of the lessons apparently take more time than anticipated because students take notes. For long lessons, a number of

alternatives may be feasible: allow 2 sessions to complete a lesson, or provide handouts of the text parts of lessons and tell students to read them before coming to PLATO, and to spend PLATO time answering questions or doing exercises.

A decision has to be made about the specific goal of each lesson. If a lesson is a list of a lot of rules, perhaps a handout or a textbook reference should be used, and PLATO should provide practice with help sequences and corrective information when the student gives an incorrect response. If teaching a concept is the goal of the lesson, concrete examples should be helpful along with the abstract ideas. Some of the lessons develop ideas logically for the person who already understands the concept. However, from the point of view of the "naive" learner, a different sequence or a particular emphasis is warranted. Considerable benefit might be derived by revision of such lessons under the guidance of a first-rate lecturer who has insights with respect to such problems. An important step in lesson revision or writing is for the author to observe a student (or a person who is not one of his colleagues) do the lesson. This should provide considerable information about the kind of responses to expect and the kind of difficulties the student will have in understanding. Lessons should be flexible in response judging, allowing for a broad span of correct alternatives.

Students like to have self-quizzes or sets of practice exercises available. A good paradigm might be to require that a minimum number of problems be done correctly, with the option to do as many more as the student chooses.

4. Student attitudes were generally favorable to CS 105 on PLATO. There was apparently no performance decrement compared to previous semesters. The task now is to make revisions so as to reach an even larger proportion of the students.

8.3. Fall, 1975

After a year's experience using the FORTRAN lessons, and with some revisions planned for the summer of 1975, it was decided to conduct a large scale, controlled experiment in CS 105, in the fall of 1975, in order to determine the effectiveness of the lessons. In order to control for effects of instructor and time of day, four CS 105 lecture sections were scheduled, two at 9:00 am and two at 10:00 am. Students at each hour were randomly assigned to one of the two sections, and one section at each hour was arbitrarily chosen to use PLATO to replace one lecture per week, while the other had two lectures. Finally, two professors (A and B), neither of whom had ever used PLATO prior to the start of classes, were assigned to the sections so that professor A taught a PLATO section at 9:00 and a non-PLATO section at 10:00, while professor B did the reverse. (A fifth section, taught by a third professor, used PLATO, but was not involved in the experiment.) More details concerning this experiment are available in Montanelli (1976).

The three hypotheses of this study were:

1. PLATO students would enjoy the course more, and give it a stronger recommendation to their friends.
2. PLATO and non-PLATO students would perform equally well on exams and homeworks in the course.
3. The drop rates in the two types of sections would be similar.

In answer to the question (from the questionnaire administered with the final exam): 'If a friend were taking CS 105 next spring and PLATO and non-PLATO sections were offered, what would you recommend he take?' PLATO students strongly recommended PLATO (112 circled 'definitely

PLATO', 88 'PLATO if convenient', 45 had 'no recommendation', 15 said 'lecture if convenient', and 21 said 'definitely lecture'. On the other hand, non-PLATO students were neutral (their responses, in order, were 29, 22, 91, 26, 19), or even showed a slight preference for PLATO.

In order to compare learning across groups, a 2x2 univariate analysis of variance was computed for each exam and for total points on computer programs. No significant differences were found, and means were nearly identical for the various groups.

The third hypothesis concerning drop rates was rejected, however. Professor A had 19 (15%) drops from his PLATO section, and only 4 (4%) from non-PLATO. Professor B had 28 (25%) drops from PLATO, and 18 (14%) from non-PLATO.

Students in the PLATO groups would strongly recommend that their friends choose PLATO sections, thus confirming the first hypothesis. Even if the 'extra' 25 drops in the PLATO sections were strongly negative, they would have a small effect on the totals of 200 PLATO students recommending PLATO, and only 36 of them recommending lectures. It should be remarked that when the PLATO students were asked to indicate what they thought were the worst features of PLATO, 78 checked 'The distance to CERL' (Unfortunately the terminals are located on the north edge of campus in CERL, about a mile from most commerce courses.), while 37 checked 'Lack of human contact', and 31 checked 'PLATO going down', the next two most frequently checked responses. Thus the major problem was unfortunately out of our control.

The second hypothesis was not rejected, due to nearly identical scores on exams and machine problems. There is no reason to suspect that the PLATO drops were poor students. However, if the dropped PLATO students were below average, they could not have had a large enough effect on the results to alter the obvious conclusion. This result is certainly in agreement with most studies of the effects of CAI. In fact, when Jamison, Suppes, and Wells (1974) surveyed the effectiveness of alternative instructional media, they stated:

> '... the equal-effectiveness conclusion seems to be broadly correct for most alternate methods of instruction at the college level ...',

and suggested studying costs of various methods of delivery. However, a major advantage of CAI is that once used, it is not set in stone like a textbook or movie. As a result of this experiment, the two lessons which students liked the least were rewritten from scratch. Secondly, a quiz system has been begun. When completed, it will present a quiz to each student at the completion of each lesson. The quizzes are not written by the authors of the lessons, and in fact quiz authors are discouraged from looking at the lessons. However, the quizzes are written from the same objectives that were used to write the lessons. The resulting quiz scores will not only tell the students how well they understand the material which the lesson is *supposed* to cover, but will tell instructors and lesson authors how well the lesson is working. Thus, continual improvement is possible, and perhaps eventually CAI materials will be as good as the best lecturer, and therefore better than many.

On the other hand, the hypothesis about equal drop rates was rejected. This was a surprising result, especially when the smaller experiment a year earlier (under worse conditions) showed no differences. However, the earlier course may have been a special case. It was a relatively small, elective course with mainly juniors and seniors in psychology and similar fields. These students were more involved and interested in the experiment, and they may have stayed for that reason. On the other hand, CS 105 is a required course for freshmen in the college of commerce, and the students were presumably less interested in long term educational goals (for themselves as well as for the PLATO materials). However, although this drop rate was disturbing, there were a few, likely reasons for it, all of which could be fixed. For one thing, the first three weeks were confusing for the students because they had pre-enrolled in a course which they expected would consist of two lectures and a discussion each week. Instead, three-fifths of them had a lecture cancelled and had to sign up for a PLATO section instead. These sections caused a lot of trouble, as some were scheduled for week-ends, and many students complained that they were unable to meet any of the remaining available PLATO times. Although most of this confusion was necessary due to the nature of the experiment, in the future students will preregister for PLATO sections just as for any other class. A second possible cause for the different drop rates was that for the first few weeks, PLATO students were required to do their programming problems in one of the online, interactive compilers. Although it was thought that this would be fun for the students, the compiler gave very poor response time because of the amount of processing going on to check for errors after each student keypress. Finally, drops might have been due in part to student dissatisfaction with the two poor lessons which were later rewritten. Students had not been systematically polled about the lessons before, and the relatively negative reaction to two of them was quite surprising.

Another possible explanation for the higher drop rate on PLATO, is that some students (< 10%) are anti-machine and that CAI will always have this problem. The authors do not feel that the large differences found here could be attributed to this reason. However, some data on this question is reported below, for CS 105 in spring 1976.

In summary, this experiment showed that PLATO lessons can be used to replace one lecture a week in an introductory computer programming course. Students learned as much and prefered PLATO to large lecture sections. The remaining problems are: 1) Is there a higher drop rate on PLATO? and 2) Can instruction be improved through continued development of the CAI materials?

At the same time as the large experiment was being conducted in CS 105, some smaller studies were being conducted with the CS 103 class. In the first of these, Barber's (1975) lesson on FORTRAN arithmetic was used to test student attitude and performance differences after using different versions of the lesson. Each of the 81 students was randomly assigned to 1 of 6 experimental groups and studied the lesson in one of three modes (PLATO coercive, PLATO non-coercive, handouts) and under one of two conditions of questioning (more or less frequent). PLATO coercive students were forced to do the exercises in the lesson, while the PLATO non-coercive group could use a key marked ANS to obtain the answers to the online exercises, without actually trying them. Handout students were locked out of PLATO and were given handouts which essentially consisted of copies of the displays the other students saw at the terminal. The "more frequent questioning" groups had additional exercises placed in various parts of the text so that almost every section ended with a few exercises. The "less frequent questioning" condition was simply the original lesson which contained a few comprehensive drills in various places.

Some of the interesting results from this experiment consisted of findings of no significant differences. First, students in the non-coercive groups did not use the ANS key nor exhibit any other attitude or performance differences from the coercive groups, so they were combined for further analyses. Secondly, there were no performance differences between PLATO and handout students or between more or less frequent questioning groups, on either the quiz over FORTRAN arithmetic given several days after the lesson or on a question on FORTRAN arithmetic on an exam given three weeks later.

There were some interesting differences in attitudes found on a questionnaire distributed immediately following the PLATO session. Two-way analyses of variance, mode (PLATO vs. written) by questioning (more vs. less frequent), were run on the questionnaire items. There were significant ($F(1, 77) = 16.8$, $p < .001$) differences between PLATO and non-PLATO groups in whether the students would recommend PLATO or a handout to a friend. PLATO students recommended PLATO, while non-PLATO students were neutral. Other statistically significant results tended to show that PLATO students with less frequent questioning were less happy about some parts of the lesson than the PLATO students with more frequent questioning or those with handouts.

Special benefits that derived from student-machine interaction were in the affective domain. There were significant differences in attitudes, perception, and expectation of CAI. Many of the advantages of the PLATO presentation were also available in the handouts, but only the PLATO students perceived them as such. Among those cited were feedback, active participation, and self-quizzes. All of these were also available in the handout but were not mentioned as advantages by those students.

The only difference between the two media was that PLATO judged and commented on each answer, but the handout student had to look up answers and comments in the back. The PLATO version was perceived to be fun, entertaining, and more interesting by 13 of 45 students who responded and was credited with making it easier to learn by 11 students. They apparently had different expectations of CAI than a textbook. They complained that "you can't ask it questions." The same was true of the handout, but nobody expected it from a text.

More frequent questions on PLATO gave the students more confidence that they had learned the material and a stronger feeling that the feedback helped their understanding. There was no evidence of such differences for the handout students in the two questioning conditions.

Student behavior was essentially the same in the PLATO coercive and non-coercive conditions. Contrary to expectation (Barber, 1975; Anderson & Faust, 1973), students under learner control engaged in appropriate learning strategies. The coercive students did not balk at the requirements that were imposed. The attitudes evidenced on the questionnaire were consistent with this behavior. Students did not have a strong feeling that they prefered to make their own decisions about how many problems to do, nor did they prefer to think answers instead of writing them. It may be that if material is well presented, the questions are relevant, and the minimum requirements are reasonable, learner control (in the sense of coerciveness) is not an important issue. However, this hypothesis should be tested further with other students, with more complex content, and at different times in the semester.

Finally, the absence of significant performance differences between the groups on the quiz may have been due to a ceiling effect. A related explanation is that with such simple material and announced tests the students were motivated to learn the material regardless of the method of presentation. More complex content might have resulted in perf[illegible]ance differences.

The second experiment in CS 103 involved the hypothesis that students would do more work in the lessons and thus achieve better understanding of the course material if they were required to do the lessons. In order to test this hypothesis, the students in CS 103 were randomly divided into three groups. In Group 1, doing the PLATO lessons was not counted as part of the students' grades. In the other two groups, doing the lessons counted 5% (Group 2) and 15% (Group 3) of the grade with other factors down weighted accordingly. The expectation was that increasing the degree to which the lessons counted in a grade would increase studying the lessons, and thus increase learning.

Based on 80 students (26 or 27 per group) who remained in the course for at least 5 weeks and took the first exam, the number of PLATO lessons completed (using time in lesson as a completion criterion) is presented in the first row of Table 5. Although there is about 1 chance in 10 that this result could be due to chance, the fact that the numbers of lessons completed increased with the amount PLATO contributed to the grade, does support the theory that students were more likely to do the lessons if the lessons were required. If dropped students are ignored, the means showed the same trend with a similar level of statistical significance. The 8.5 may even be slightly high, due to a differential drop rate. Whereas groups 1, 2, and 3 began with 26, 27, and 27 students

Table 5

Means on Performance Data of Three Groups with Varying Percentages of Their Grades Determined from Completing PLATO Lessons

	Group 1 0%	Group 2 5%	Group 3 15%	Probability[1]
		Means[2]		
Number of PLATO lessons completed	7.7	8.5	9.7	.095
Number of PLATO lessons completed ignoring drops	8.5(20)	9.3(24)	10.5(23)	.094
Total machine problem points	136.	140.	154.	.50
Hour exam 1 written	65.7	64.2	70.9	.29
Hour exam 1 PLATO	64.7(23)	60.6(25)	72.8(25)	.006
Hour exam 2	50.8(16)	50.0(23)	52.6(23)	.84
Final exam	128.3(18)	114.6(23)	129.3(23)	.17

[1] Probability of the observed F value from a 1-way analysis of variance between the 3 groups.

[2] Group sizes were 26, 27, and 27, unless specifically indicated. (i.e. 16 students in Group 1 took the second exam).

respectively, they finished with 20, 24, and 23. Performance data generally favored group 3 (PLATO 15% of grade), although there were either no differences between groups 1 and 2, or group 1 did better than group 2. The only statistically significant result indicated that students in group 3 did better on the part of the first hour exam which was given on PLATO.

Several other kinds of data were also collected in CS 103 in the fall of 1975. Table 6 presents some data on student usage of the required PLATO lessons. Column 1 gives the date on which the data was taken. In general, lessons were required to be completed by the end of the week following that during which the assignment was made, but all previously assigned lessons were required to be done before an exam, which accounts for the pile-up around September 29. Column 2 gives names (somewhat mneumonic) of the required lessons. Column 3 reports the number of occurrences of a type of error which occurred when a lesson did not properly return to the operating system after execution. Lesson fortchar (FORTRAN character handling) had a relatively large number of such "errors" due to an experimental quiz which was appended to it (see section 5.4. on the quiz system). Column 4 gives the number of times each lesson was invoked by a student in the class, and column 5 gives the number of students trying the lesson. Multiple entries to one lesson by one student may be due to review, failure to finish in one sitting, mistaken exit from a lesson, system failure, and possibly other reasons. Column 6 gives the number of students who completed (executed the last piece of code in every section and subsection) the lesson. Column 7 gives the average amount of time

Table 6

CS 103 Fall 1975 - Lesson Data Accumulated Online

Date	Lesson	Bad exits	Number of uses	Number of students entered	Number of students completed	Average time per student	Average time per student completing
9/8	csintro	2	129	84	81	31	32
9/8	fortintro	1	134	60	49	26	30
9/8	fortarith	6	197	89	48	47	80
9/15	fortif	4	214	76	70	42	45
9/25	fortfmt1	7	272	82	74	68	73
9/29	fortarray1	5	177	81	60	73	86
9/29	fmtsim	0	208	66	44	42	56
10/20	fortarray2	3	87	48	22	47	66
10/27	binsearch	1	91	52	39	27	33
11/3	fortfmt2	2	149	57	53	64	68
11/3	numbers	5	160	46	37	63	75
11/16	fortsub1	0	94	49	26	49	70
12/2	fortsubex	1	95	55	18	30	53
12/4	fortchar	19	122	57	34	32	46

(in minutes) spent in the lesson by each student, and column 8 gives the average time for students who completed the lesson. Thus, the last column in Table 6 gives upper bounds (because some students were reviewing) for the average times to complete the lessons. Note that the relatively poor percentage of students finishing fortarith is an artifact due to the experiment described earlier in which one-third of the students signed-on to the system, but were directed to a handout and prevented from finishing the lesson. Although the generally decreasing numbers in columns 5 and 6 would seem to indicate decreasing use of and interest in PLATO, they are also due to a relatively high drop rate which left only 67 students finishing the course after an initial enrollment of 87. This result was largely due to 13 students who dropped after taking the first exam (in the 6th week), but before the 8th week deadline. One-half of this exam was given on PLATO, and there were several problems resulting in lost exams and frustrated students which could have caused some of the extra drops. Another indication of dissatisfaction with the PLATO exam came from student responses to an end of semester questionnaire. Although only a few students criticized PLATO, four said it was fine except for the exam. Also near the end of each semester, each computer science course administers a course evaluation questionnaire through which students rate various aspects of each course. A weighted sum of all the questions is used as an overall evaluation, and this course received a 2.7 (on a scale from 1 to a top of 4). In the fall of 1974 when one-half of the course was on PLATO, the rating was 2.95, and in the fall of 1973, before PLATO, the result was 2.99, all with the same instructor.

8.4. Spring, 1976

Although no experiments were run, certain data was collected in CS 105 in order to observe routine use of PLATO. For example, Table 7 presents course evaluation results for 6 semesters in CS 105. Only those professors who taught at least two sections, with at least one on PLATO are included. Although results within the fall 1975 semester (professors E and F) would tend to be interpreted as showing lower evaluations for professors in PLATO sections as compared with those same professors in non-PLATO sections, results from the three professors (A, B, and C) who taught PLATO and non-PLATO sections in separate semesters do not support this idea. One possible explanation is that both professors E and F remarked after fall 1975, that they tried to give the same first lecture to both PLATO and non-PLATO groups. They both felt that their lectures to the PLATO sections lacked continuity and often overlapped or left gaps with the PLATO lessons.

8.5. Summary

A few general conclusions can be drawn from the large amount of data analyzed here. First, PLATO lessons originally written by students can be used to replace one lecture per week in introductory computer science courses without decreasing student learning. Secondly, students who finish a course using PLATO are happier with the course than non-PLATO students, and would recommend that fellow students choose PLATO sections. However, there is apparently a somewhat higher drop rate in PLATO sections which has withstood various modifications and improvements. Finally, the compilers and the exam system are not yet ready for routine student use.

Table 7

CS 105 Course Evaluation Questionnaire Results

	Pre-PLATO			PLATO			PLATO
	Fall 1973	Spring 1974	Fall 1974	Spring 1975	Fall 1975		Spring 1976
Professor					PLATO	non-PLATO	
A	3.00				2.93		
B		2.64 2.67					2.63 2.58
C			2.80 2.80	2.84			
D				2.68 2.67			2.55
E					2.49	2.56	
F					2.56	2.75	2.34 2.48

References

[1] Anderson, R. C. and Faust, G. W. Educational Psychology, Dodd, Mead and Company, New York, 1973.

[2] Barber, J. A. Data collection as an improvement technique for PLATO lessons. Report UIUCDCS-R-75-777 (M.S. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, December 1975.

[3] Montanelli, R. G., Jr. CS 103 PLATO experiment, Fall 1974. Report UIUCDCS-R-75-746, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1975.

[4] Montanelli, R. G., Jr. Evaluation of the use of CAI materials in an introductory computer science course. presented at the AEDS International Convention, Phoenix, Arizona, May 1976.

[5] Jamison, D., Suppes, P. and Wells, S. The effectiveness of alternative instructional media: a survey. Review of Educational Research, Vol. 44, No. 1, Winter 1974, 1-67.

9. ACSES Bibliography

Anderson, R. I. User's manual and guide to the ACSES quiz system. to appear as DCS Report, September 1976.

Anderson, R. I. An experiment on modes of question answering. to appear as DCS Report, December 1976.

Barber, J. A. Data collection as an improvement technique for PLATO lessons. Report UIUCDCS-R-75-777 (M.S. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, December 1975.

Barnett, R. D. An interactive COBOL system for PLATO. Report UIUCDCS-R-75-685 (M.S. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, January 1975.

Danielson, R. and Nievergelt, J. (1975). An automatic tutor for introductory programming students. Proc. Fifth Symp. on Computer Science Education, SIGCSE Bulletin, Vol. 7, No. 1, February 1975.

Danielson, R. L. PATTIE: An automated tutor for top-down programming. Report UIUCDCS-R-75-753 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, October 1975.

Davis, A., Tindall, M. H. and Wilcox, T. R. (1975). Interactive error diagnostics for an instructional programming system. Proc. Fifth Symp. on Computer Science Education, SIGCSE Bulletin, Vol. 7, No. 1, February 1975.

Davis, A. M. An interactive analysis system for execution-time errors. Report UIUCDCS-R-75-695 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, January 1975.

Eland, D. R. An information and advising system for an automated introductory computer science course. Report UIUCDCS-R-75-738 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, June 1975.

Embley, D. W. An experiment on a unified control construct. Report UIUCDCS-R-75-759, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1975.

Embley, D. W. and Hansen, W. J. The KAIL selector - a unified control construct, SIGPLAN Notices, Vol. 11, No. 1, January 1976.

Embley, D. W. An experiment on CAI sequencing constructs. Report UIUCDCS-R-76-771, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1976.

Embley, D. W. Experimental and formal language design applied to control constructs for interactive computing. Report UIUCDCS-R-76-811 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, July 1976.

Gillett, W. D. An interactive program advising system. Proc. of SIGCSE-SIGCUE Joint Symp. on Computer Science Education, SIGCSE Bulletin, Vol. 8, No. 1, February 1976.

Gillett, W. D. Iterative techniques for detection of program anomalies. submitted to the Conference on Principles of Programming Languages, Los Angeles, California, January 1977.

Gillett, W. D. Interval maintenance in an interactive environment. in preparation.

Izquierdo, F. J. A generator/grader of problems about syntax of programming languages to be used in an automated exam system. Report UIUCDCS-R-75-755 (M.S. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, September 1975.

Mateti, P. An automatic verifier for a class of sorting programs. (Ph.D. Thesis), to appear as DCS Report, September 1976.

Montanelli, R. G., Jr. CS 103 PLATO experiment, Fall 1974. Report UIUCDCS-R-75-746, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1975.

Montanelli, R. G., Jr. Evaluation of the use of CAI materials in an introductory computer science course. presented at the AEDS International Convention, Phoenix, Arizona, May 1976.

Montanelli, R. G., Jr. Using CAI to teach introductory computer programming. submitted to Communications of the ACM.

Montanelli, R. G., Jr. and Steinberg, E. R. Using PLATO to teach introductory computer science - an overall evaluation. in preparation.

Nakamura, S. Reorganization of an interactive compiler. (M.S Thesis), to appear as DCS Report, August 1976.

Nievergelt, J., Reingold, E. M. and Wilcox, T. R. The automation of introductory computer science courses. in A. Gunther, et al. (eds), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

Nievergelt, J., et al. ACSES, an automated computer science education system. Angewandte Informatik, Vol. 3, April 1975.

Nievergelt, J. Interactive systems for education - the new look of CAI. Invited paper presented at the IFIP World Conf. on Computer Education, Marseille, France, September 1975.

Pradels, J. L. The GUIDE, an information system. Report UIUCDCS-R-74-626 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, March 1974.

Renshaw, J. W. Robocar: educating the layman in computer science. Report UIUCDCS-R-75-741 (M.S. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, July 1975.

Segal, B. Z. A comparison of student performance under two methods of error announcement. Report UIUCDCS-R-75-727 (M.S. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, May 1975.

Steinberg, E. R. and Montanelli, R. G., Jr. Effects of coerciveness and aspects of human-machine interaction in a computer science CAI lesson. to be submitted to Journal of Computer-based Instruction.

Tindall, M. H. An interactive table-driven parser system. Report UIUCDCS-R-75-745 (M.S. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, August 1975.

Tindall, M. H. An interactive compile-time diagnostic system. Report UIUCDCS-R-75-748 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, October 1975.

White, L. A. CAPS compiler CPU use report. Report UIUCDCS-R-75-790, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1975.

Whitlock, L. R. Interactive test construction and administration in the generative exam system. Report UIUCDCS-R-76-821 (Ph.D. Thesis), Department of Computer Science, University of Illinois at Urbana-Champaign, September 1976.

Wilcox, T. R. The interactive compiler as a consultant in the computer aided instruction of programming. Proc. of the Seventh Annual Princeton Conference on Information Sciences and Systems, March 1973.

Wilcox, T. R., Davis, A. and Tindall, M. H. The design and implementation of a table-driven, interactive diagnostic programming system. to appear in Communications of the ACM.

Wilcox, T. R. An interactive table-driven diagnostic editor for high-level programming languages. in preparation.

Appendix: Computer Science Lessons

a. Sequencing and Entry Lessons

Name	Description	Status
cs	Entry into the ACSES System	operational
csguide	Conversational Request Translator and Processor	operational
cslessons	Master Index to the Computer Science Lessons	operational
introprog	Introduction to the Mini-Language Sequence	just started
introprogc	Intro. to Language Independent Programming Sequence	nearly complete
pllintro	Introduction to the PL/1 Lesson Sequence	operational
fortintro	Introduction to the FORTRAN Language and Lessons	work in progress
basicintro	Introduction to the BASIC Lesson Sequence	just started
cobolintro	Introduction to the COBOL Lesson Sequence	operational
aplintro	Introduction to the APL Sequence	operational
logointro	Introduction to the LOGO Lesson Sequence	just started
introstrct	Introduction to the Data Structures Sequence	operational
intronum	Introduction to the Numerical Analysis Sequence	just started
intrologic	Introduction to the Logical Design Sequence	operational
csrouter	Router Lesson for Computer Science Courses	operational

b. General and Miscellaneous

Name	Description	Status
csintro	Introduction to Computers and Computer Programming	work in progress
algolingo	Introduction to Algorithms	operational
numbers	Number Representation in Computers	work in progress
hp45sim	HP45 Calculator Simulator	nearly complete
epic	Simulation of Epic 2000 Calculator	operational
turing	Turing Machines	nearly complete
darwinl	Programming War Game	work in progress
compchess	Computer Chess Programming Techniques	nearly complete
mazesearch	Maze Traversing Algorithm	nearly complete
graphix	Manual for Grafix	work in progress
platoiv	Plato Hardware and Software	work in progress

c. Mini-Languages

Name	Description	Status
introprog	Introduction to the Mini-Language Sequence	just started
pal	Pictorial Programming Language for Children	nearly complete
somaga	Drawing Language	operational
pl2d	Recursion	operational
csmini	Mini Programming System Prototype	operational
cstrees	Tree and List Manipulation Mini-Language	operational
roboint	Introduction to the Robot Car Sequence	work in progress
robocar	Robot Car Mini-Language	work in progress
robostack	Robot Car Stack Algorithm	work in progress
roboback	Robot Car Backtrack Algorithm	work in progress

d. Language Independent Programming

Name	Description	Status
introprogc	Introduction to Language Independent Programming Sequence	nearly complete
flowchrt	Flow Charting	nearly complete
loops	DO-Type Loops	operational
beginblock	Begin Blocks	operational
detab	Decision Tables	operational
files	File Processing	operational
recurse	Recursion	operational
trigraf	Directed Development of a Program	work in progress
formlang	Formal Computer Languages	nearly complete
vwgrammar	Two Level Grammars	work in progress

e. PL/1 Language

Name	Description	Status
pllintro	Introduction to the PL/1 Lesson Sequence	operational
pllarith	PL/1 Arithmetic Operations	operational
pllstring	String Operations in PL/1	operational
pllif	PL/1 IF Statements and DO Groups	operational
plldo	PL/1 DO Stetments	operational
pllarray	PL/1 Arrays	nearly complete
pllarrayx	Advanced Examples of PL/1 Arrays	nearly complete
pllproc	PL/1 Procedures and Subprograms	nearly complete
pllio	PL/1 LIST Input/Output	operational
plledit	PL/1 EDIT Input/Output	nearly complete
plleditdrl	PL/1 EDIT Input/Output Drill	nearly complete
pllpic	PL/1 PICTURE Specification	work in progress
pllrecurse	PL/1 Recursive Procedures	nearly complete
pllstrl	Data Structures in PL/1	operational

f. FORTRAN Language

Name	Description	Status
fortintro	Introduction to the FORTRAN Language and Lessons	work in progress
fortarith	Introduction to FORTRAN Arithmetic	operational
fortif	FORTRAN IF Statements	operational
fortdo	FORTRAN DO Loops	nearly complete
fortarray1	One Dimensional Arrays	operational
fortarray2	Two Dimensional Arrays	operational
fortsub1	FORTRAN SUBROUTINE Subprograms	operational
fortsubex	FORTRAN SUBROUTINE Examples	operational
fortfunct	FORTRAN FUNCTION Subprograms	just started
fortfmt1	Introduction to FORTRAN FORMAT Statements	operational
fortfmt2	Advanced FORTRAN FORMAT Statement	operational
fmtsim	FORTRAN FORMAT Simulator	nearly complete
fortchar	Character Handling in FORTRAN	operational

g. BASIC Language

Name	Description	Status
basicintro	Introduction to the BASIC Lesson Sequence	just started
basicbasic	Introductory BASIC	just started
basicref	Beginning BASIC	work in progress
basicref1	Advanced BASIC	work in progress
basicloop	FOR-NEXT Loops in BASIC	just started
basicarray	Arrays in BASIC	work in progress

h. COBOL Language

Name	Description	Status
cobolintro	Introduction to the COBOL Lesson Sequence	operational
coboliden	COBOL Identification and Environment Divisions	operational
coboledit	Advanced COBOL PICTURE Clauses	operational
coboldata	COBOL Data Division	operational
cobolproc	COBOL Procedure Division	operational
cobolref	COBOL Language Reference	work in progress

i. APL Language

Name	Description	Status
aplintro	Introduction to the APL Sequence	operational
aplscalar	APL Scalars	operational
aplvector	APL Vectors	operational

j. Machine and Assembler Language and Computer Simulators

Name	Description	Status
minie	A Simple Computer	operational
machlang	Machine Language	work in progress
pdp8sim	PDP8/L Simulator	work in progress

k. Other Languages

Name	Description	Status
snobol	SNOBOL4	revision needed
lisp	LISP List Processing Language	work in progress
logointro	Introduction to the LOGO Lesson Sequence	just started
logotest	LOGO Test Instructions	just started
logoproc	LOGO Procedures	just started
logocom	LOGO Commands	work in progress

1. Information Processing

Name	Description	Status
sortintro	Introduction to Sorting	work in progress
sorting	Sorting	revision needed
sortlab	Sort Program Judging	work in progress
binsearch	Binary Searching	work in progress
binsearch1	Binary Searching with FORTRAN and PL/C	nearly complete
introstrct	Introduction to the Data Structures Sequence	operational
str1	Information Structures	operational
str2	Information Structure Drills	operational
str3	Experience with Stacks	operational
lister	Experience with List Space	work in progress
noder	Experience with List Nodes	operational
treetrav	Traversal of Binary Trees	operational
cstrees	Tree and List Manipulation Mini-Language	operational

m. Numerical Analysis

Name	Description	Status
intronum	Introduction to the Numerical Analysis Sequence	just started
matmult	Matrix Multiplication	work in progress
numquad	Numerical Integration	revision needed
lineq1	Linear Equations	work in progress
lineq2	Linear Equations	revision needed
rootlab	Non-Linear Equations	revision needed
leastsq	Least Squares	operational
linprog	Linear Programming	operational
montecarlo	Monte Carlo Methods	operational
splines	Spline Approximation	work in progress

n. Applications

Name	Description	Status
simulation	Discrete Simulation	work in progress
traficsim	Traffic Simulation	operational
racetrack	Simulation Games	operational
payroll	Payroll Program	operstional
csslides	Computer Uses in Business	work in progress

o. System Programming

Name	Description	Status
semaphore	Experience with Dijkstra Semaphores	operational
deadlock	Illustration of the Deadlock Problem	operational
buffering	Experience with I/O Supervisor Buffering Routines	operational
lexi	Finite State Machine for Lexical Analysis	operational
topdown	Top-Down Syntax Analysis	operational
stackalotl	Bottom-Up Analysis of Expressions	nearly complete
coder	Code Generation by Templates	operational

p. Computing Services Office

Name	Description	Status
csointro	Introduction to UIUC Computing Services Office	work in progress
jclbasic	IBM OS/360/370 Job Control Language	work in progress
csodata	IBM 360 Load Modules and DEC-10 SAV Files	work in progress
calcomp	CalComp Plotter	operational
online	Remote Terminals	work in progress

q. Logical Design

Name	Description	Status
intrologic	Introduction to the Logical Design Sequence	operational
logicarith	Introduction to Digital Arithmetic	operational
logicgate	Combinational Building Blocks	operational
logicmin	Minimization of Boolean Expressions	operational
logicff	Basic Sequential Building Blocks--Flip Flops	nearly complete
logicseq	Sequential Circuit Design	work in progress
logiccdr	Combinatorial Problems	operational
logicmsi	MSI Logical Building Blocks	operational
logichdw	Semiconductor Fabrication Methods	operational
logicflow	Data Flow Diagrams	operational
logiclab	Logic Laboratory	just started
boolex	Boolean Expressions	work in progress
logiccomb	Combinations of Logic Circuits	just started

r. Compilers

Name	Description	Status
refmanual	Reference Manual for the On-Line Compilers	operational
cursedit	FORTRAN Compiler	operational
wits	FORTRAN and BASIC Compilers	operational
pl1comp	PL/1 Compiler	under revision
pl1comp2	PL/1 Compiler with Line Editor	under revision
fortcomp	FORTRAN Compiler	under revision
fortcomp2	FORTRAN Compiler with Line Editor	under revision
basiccomp	BASIC Compiler	under revision
cobolcomp	COBOL Compiler	under revision
pascalcomp	PASCAL Compiler	under revision
snobolcomp	SNOBOL4 and SPITBOL Compiler	under revision
lispcomp	LISP Compiler	under revision

a. Communication

Name	**Description**	**Status**
cscomments	Comments between CS Students and Authors	operational
cstalk	On-Line Consultation with an Instructor	operational
csmsg	Bulletin Board for Course Messages	operational
csnotes	CS Author - Author Communication	operational

t. Lesson Writing and Evaluation

Name	Description	Status
style	Suggestions on Plato Lesson Writing Style	operational
csauthors	Useful Material and Coding Conventions for CS Authors	operational
csdesign	Graphical Lesson Structure Design	operational
csmini	Mini Programming System Prototype	operational
cslibrary	Library of Useful Routines, Charsets, Micros, Etc.	operational
cscode	Coding Suggestions for CS Lessons	operational
kail	KAIL Lesson Programming Language Compiler	work in progress
kaids	Description of KAIL Language	nearly complete
khelp	Author Aids for KAIL Compiler as Implemented	operational
csscrap	Lesson Space for Author Practice	operational
csnotes	CS Author - Author Communication	operational

BIBLIOGRAPHIC DATA SHEET

1. Report No. UIUCDCS-R-76-810

2.

3. Recipient's Accession No.

4. Title and Subtitle

ACSES: The Automated Computer Science Education System
at the University of Illinois

5. Report Date August 1976

6.

7. Author(s) J. Nievergelt, et al.

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No. NSF EC41511 and EPP-74-21590

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D.C.

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

The Automated Computer Science Educational System (ACSES) has been developed at the University of Illinois for the purpose of providing improved education for the large number of students taking introductory computer science courses. The major components of this system are: a library of instructional lessons, an interactive programming system with excellent error diagnostics, an information retrieval system, an automated exam and quis system, and several lessons which judge student programs. This report briefly describes each of these components, as well as some ideas on programming language design resulting from our experience, and presents an evaluation of the use of the system over the past three years.

17. Key Words and Document Analysis. 17a. Descriptors

computer-assisted instruction
CAI
computer science education
educational innovation
PLATO
interactive compilers
information retrieval
artificial intelligence
programming language design

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page) UNCLASSIFIED

21. No. of Pages 166

22. Price

FORM NTIS-35 (10-70)

USCOMM-DC 40329-P71